Introduction

- **Cucumber** is a tool based on **BDD** - Behavior Driven Development framework
- In **BDD** → Acceptance criteria and test cases are written in a way that simulates user behaviour.
- Cucumber is used to write and test User Stories.
- In Cucumber, User Stories are written using **Gherkin** language.
  - It is just like plain English
  - Therefore, it is **readable by Non-Technical people,** such as BA, Product Owner etc.

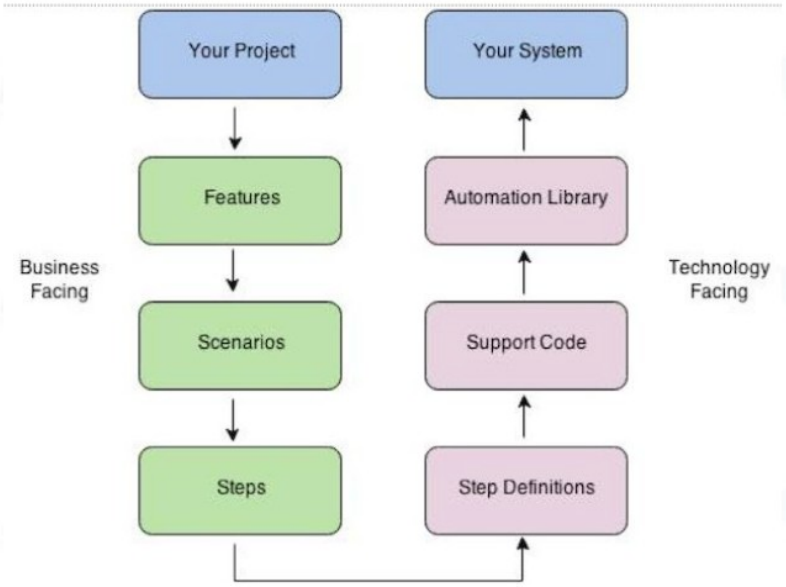
Runner Class Options

- **features** → the path of the feature files
- **glue** → the path of the step definition files
- **dryRun** → to check the mapping is proper between feature file and step def file. *(true, false)*
- **format** → to generate different types of reporting
- **monochrome** → display the console output in a more readable format. *(true, false)*
- **strict** → it will check if any step is not defined in step definition file

Step Definition

- Just because we wrote steps in Gherkin language does not mean the computer will understand and perform those tasks.
- For each and every step **we have to provide the implementations** (Actual Selenium/Java Code)
- For this we create a Java Class. It is called **step definitions** file.
  - It includes methods with glue that match exactly the step names.
  - This is how feature files gets mapped to the step definitions and get executed.

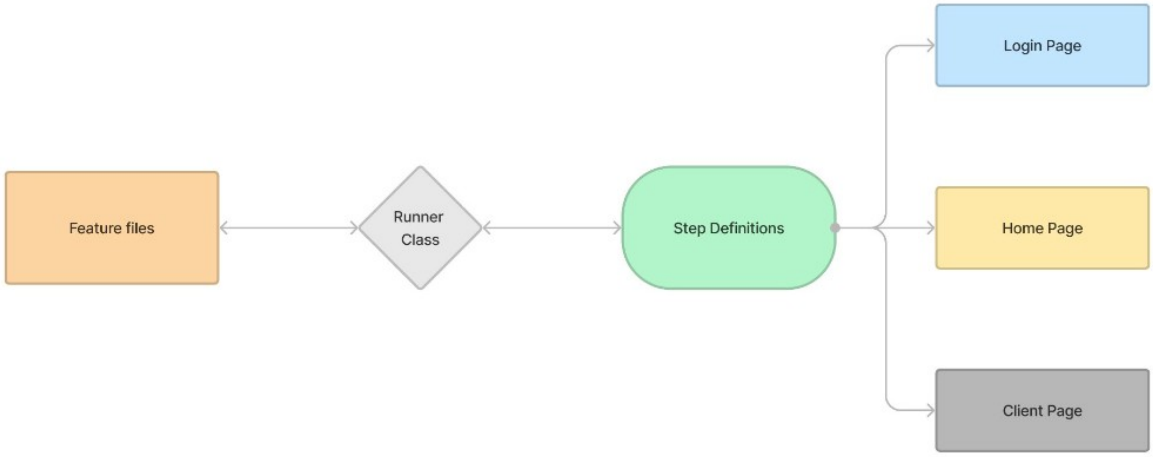
DIAGRAM



REMINDER

For Dryrun to work, change the cucumber dependencies to version: 4.8.1

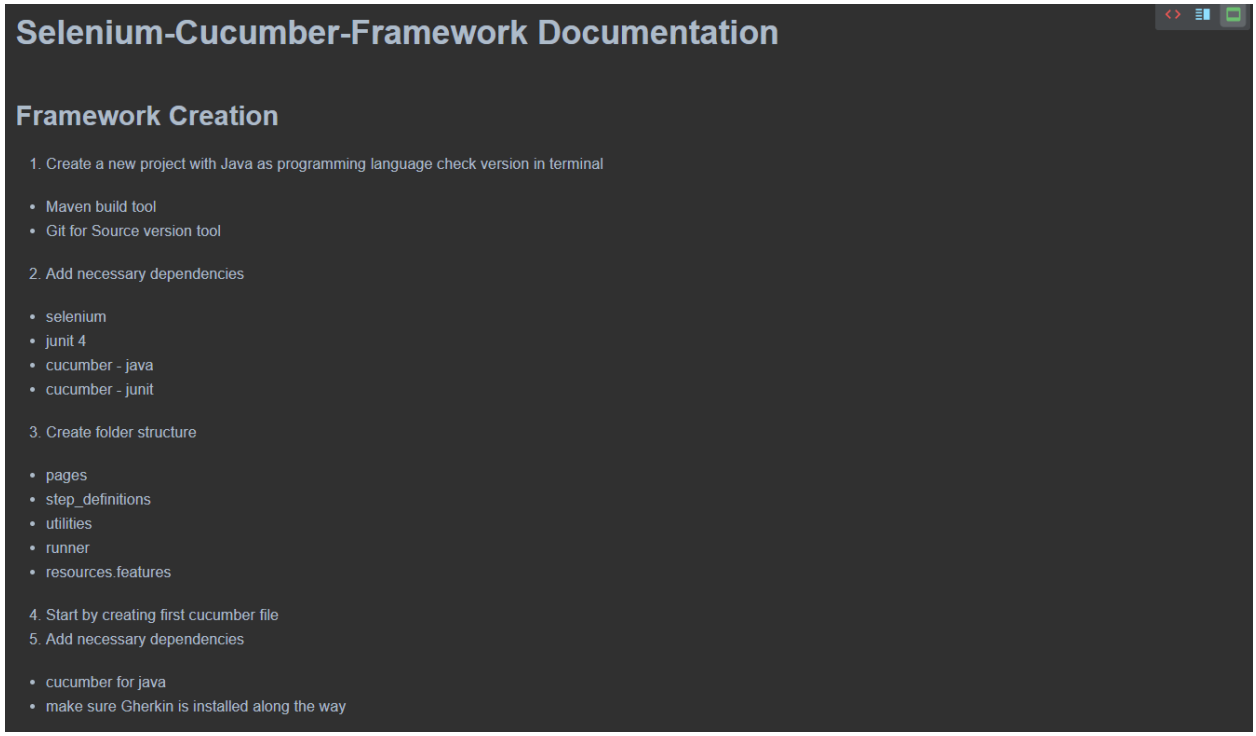
DIAGRAM



Before Starting

1. **Create :**

- A project : **Selenium-Cucumber-Demo-Framework**. Choose **git** as **repository** /**Maven** / Idcraft:**io.yoll**
- **README.md** file:
- **Write the steps, and use the small green button to preview, and the middle button to edit.**


Selenium-Cucumber-Framework Documentation
Framework Creation
1. Create a new project with Java as programming language check version in terminal
• Maven build tool
• Git for Source version tool
2. Add necessary dependencies
• selenium
• junit 4
• cucumber - java
• cucumber - junit
3. Create folder structure
• pages
• step_definitions
• utilities
• runner
• resources.features
4. Start by creating first cucumber file
5. Add necessary dependencies
• cucumber for java
• make sure Gherkin is installed along the way

2. **Add the independencies into pom.xml file.**

3. **Under** <properties> **put the version of Cucumber for junit and java:**

```
<cucumber.java.version>7.20.1</cucumber.java.version>
```

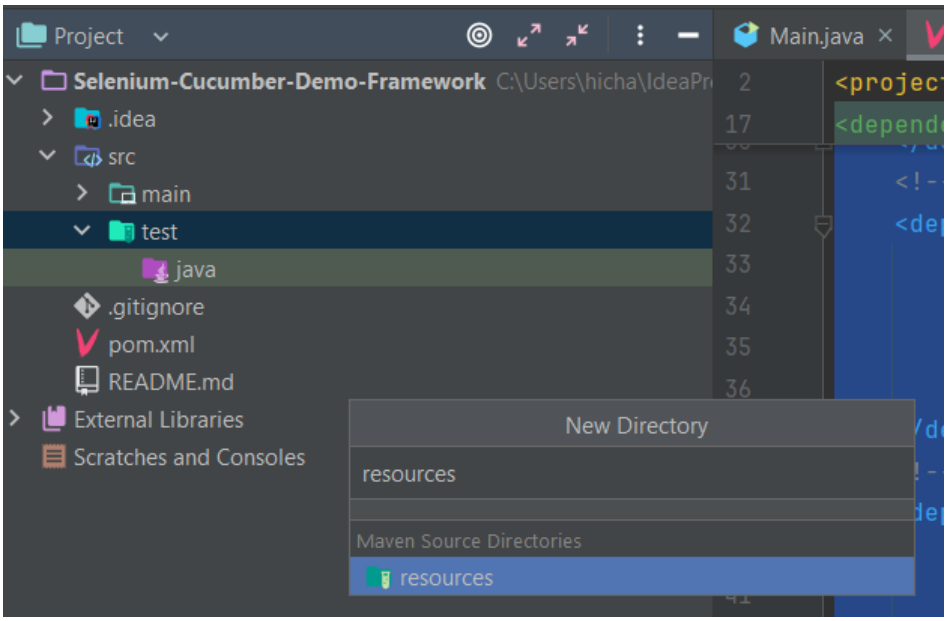
4. **On their version, put :**

```
<version>${cucumber.java.version}</version>
```

```
<properties>
        <maven.compiler.source>22</maven.compiler.source>
        <maven.compiler.target>22</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
          <cucumber.java.version>7.20.1</cucumber.java.version>
    </properties>
<dependencies>
    <!-- https://mvnrepository.com/artifact/org.seleniumhq.selenium/selenium-java -->
    <dependency>
        <groupId>org.seleniumhq.selenium</groupId>
```

```
        <artifactId>selenium-java</artifactId>
        <version>4.25.0</version>
    </dependency>

    <!-- https://mvnrepository.com/artifact/org.apache.maven.plugins/maven-compiler-
plugin -->
    <dependency>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.13.0</version>
    </dependency>
    <!-- https://mvnrepository.com/artifact/junit/junit -->
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.13.2</version>
        <scope>test</scope>
    </dependency>
    <!-- https://mvnrepository.com/artifact/io.cucumber/cucumber-java -->
    <dependency>
        <groupId>io.cucumber</groupId>
        <artifactId>cucumber-java</artifactId>
         <version>${cucumber.java.version}</version>
    </dependency>
    <!-- https://mvnrepository.com/artifact/io.cucumber/cucumber-junit -->
    <dependency>
        <groupId>io.cucumber</groupId>
        <artifactId>cucumber-junit</artifactId>
         <version>${cucumber.java.version}</version>
        <scope>test</scope>
    </dependency>

</dependencies>
```

5. **Add resources directory under test folder, it will contains everything that is NOT CODE. While java folder will contains CODING.**

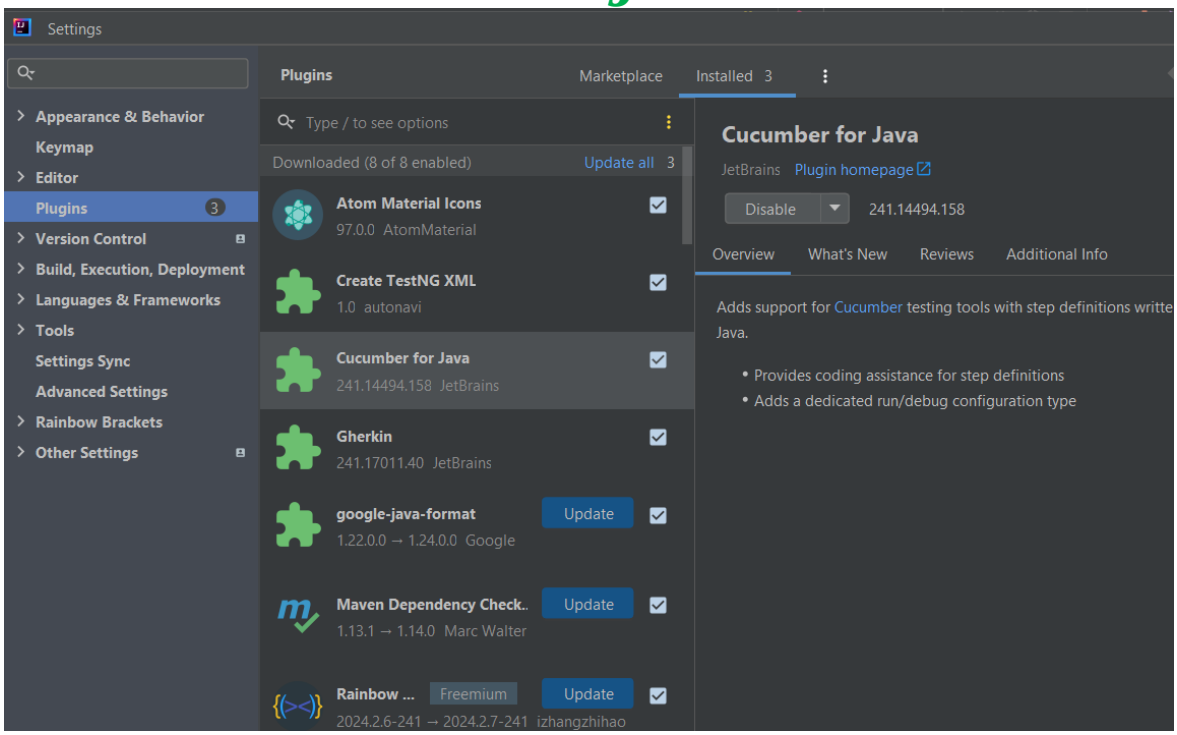


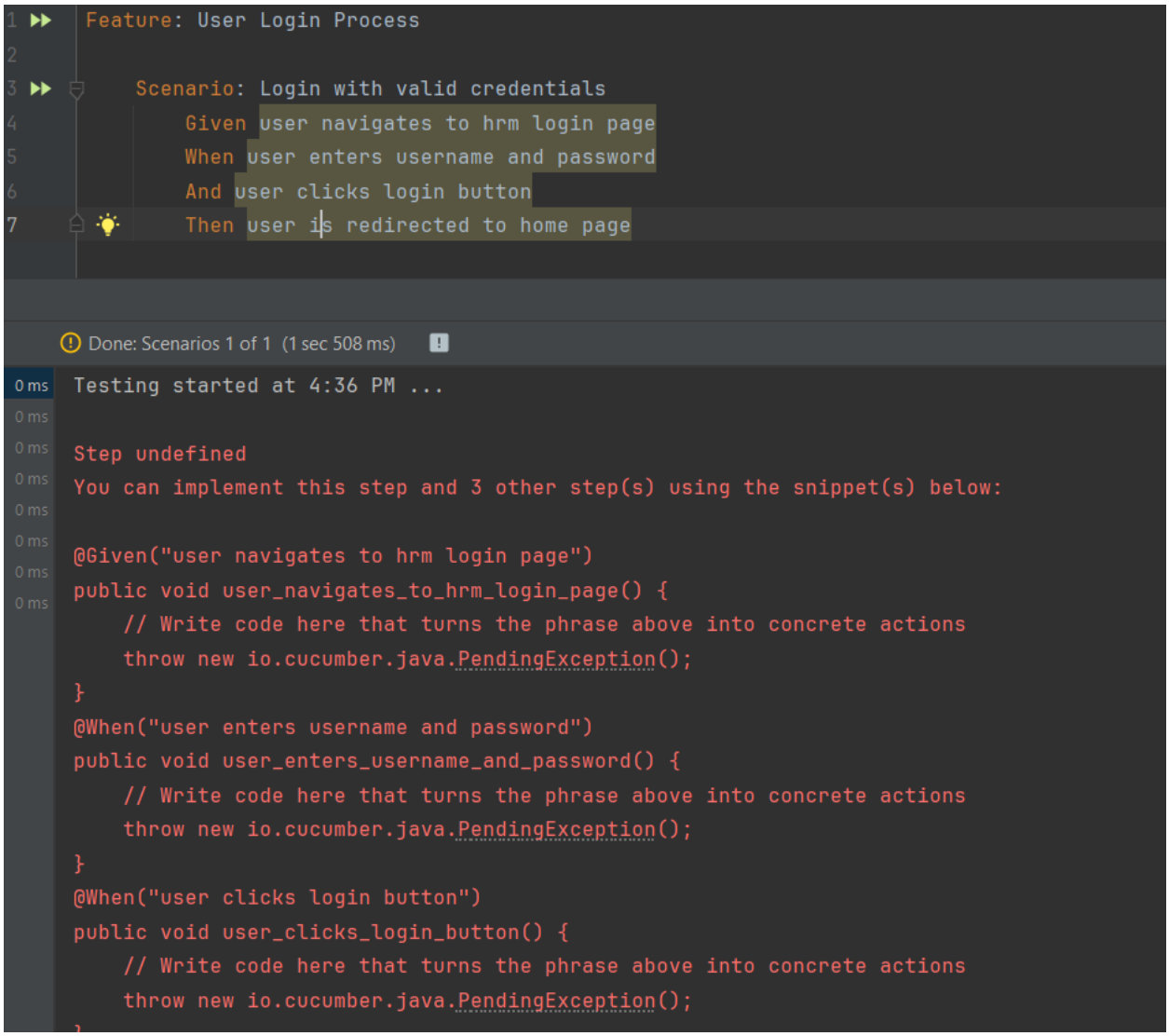
6. **Add FOUR packages under java folder : runner / pages / step_definitions / utilities**
7. **Under resources file add features Directory. It will contains CUCUMBER STUFF.**
   Then add under it : user-login.feature
   Then write :

```
Feature: User Login Process

    Scenario: Login with valid credentials
        Given user navigates to hrm login page
        When user enters username and password
        And user clicks login button
        Then user is redirected to home page
```

Then run the scenario
Then use the suggestion in red of the template that can be used.

You need to install these before running the scenario :
Go to File/ Settings/ Plugins/ Marketplace, then install : Cucumber for Java and Gherkin.





```
Feature: User Login Process

    Scenario: Login with valid credentials
        Given user navigates to hrm login page
        When user enters username and password
        And user clicks login button
        Then user is redirected to home page
```

Done: Scenarios 1 of 1 (1 sec 508 ms)

```
Testing started at 4:36 PM ...

Step undefined
You can implement this step and 3 other step(s) using the snippet(s) below:

@Given("user navigates to hrm login page")
public void user_navigates_to_hrm_login_page() {
    // Write code here that turns the phrase above into concrete actions
    throw new io.cucumber.java.PendingException();
}
@When("user enters username and password")
public void user_enters_username_and_password() {
    // Write code here that turns the phrase above into concrete actions
    throw new io.cucumber.java.PendingException();
}
@When("user clicks login button")
public void user_clicks_login_button() {
    // Write code here that turns the phrase above into concrete actions
    throw new io.cucumber.java.PendingException();
```

Putting the repeated steps on Preliminary Steps :

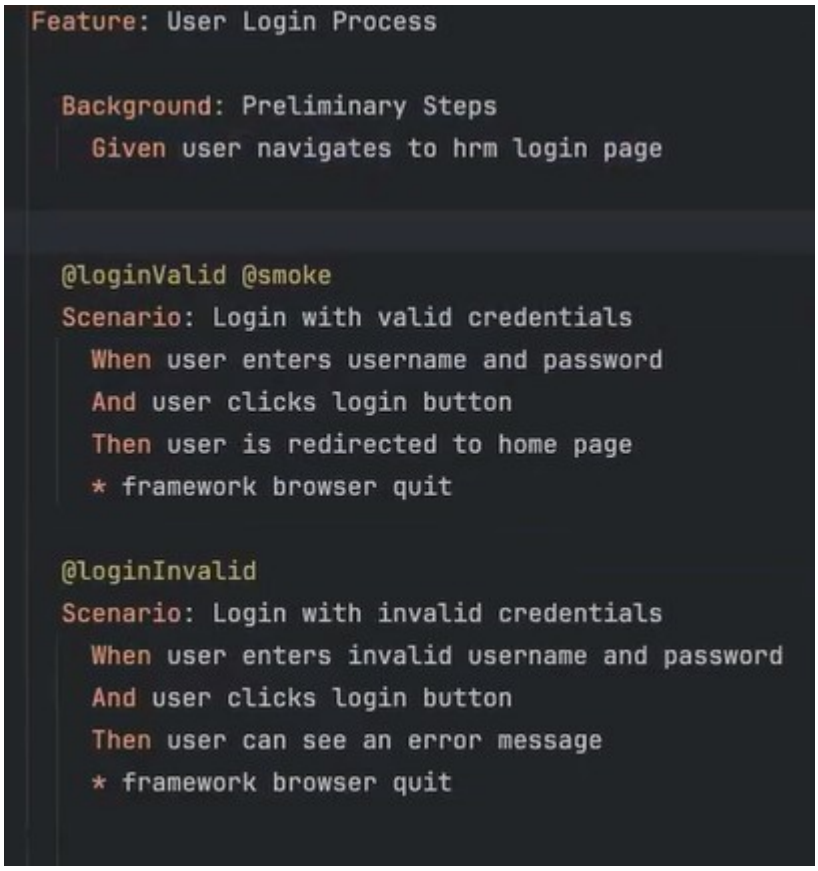
Background : Preliminary Steps

Given user navigates to hrm login page

Add on the Scenario this line:

#Given user navigates to hrm login page

```
@regression
Feature: User Login Process

  Background: Preliminary Steps
    Given user navigates to hrm login page


  @loginValid @smoke
  Scenario: Login with valid credentials
    #Given user navigates to hrm login page
    When user enters username and password
    And user clicks login button
    Then user is redirected to home page
    * framework browser quit

  @loginInvalid @smoke
  Scenario: Login with invalid credentials
    #Given user navigates to hrm login page
    When user enters invalid username and password
    And user clicks login button
    Then user can see an error message
    * framework browser quit
```



```
Feature: User Login Process

  Background: Preliminary Steps
    Given user navigates to hrm login page


  @loginValid @smoke
  Scenario: Login with valid credentials
    When user enters username and password
    And user clicks login button
    Then user is redirected to home page
    * framework browser quit

  @loginInvalid
  Scenario: Login with invalid credentials
    When user enters invalid username and password
    And user clicks login button
    Then user can see an error message
    * framework browser quit
```

```
package step_definitions;

import io.cucumber.java.en.Given;
import io.cucumber.java.en.Then;
import io.cucumber.java.en.When;

public class LoginSteps {
    @Given("user navigates to hrm login page")
    public void user_navigates_to_hrm_login_page() {
System.out.println("Given Step");
    }
    @When("user enters username and password")
    public void user_enters_username_and_password() {
        System.out.println("When Action");

    }
    @When("user clicks login button")
    public void user_clicks_login_button() {
        System.out.println("And Action");

    }
    @Then("user is redirected to home page")
    public void user_is_redirected_to_home_page() {
        System.out.println("Then Outcome");

    }

}
```

8. **Under step-definitions, Create a java class : LoginSteps, then copy/paste the red template.**
9. **Import Classes of @Given, @When, and @Then.**
10. **Delete these :**

```
    // Write code here that turns the phrase above into concrete
actions
        throw new io.cucumber.java.PendingException();
```

11. **Under runner, Create a Test Runner Java Class This class will run the Cucumber tests using Junit in order to connect user-login.feature and LoginSteps java class.**

Example: TestRunner.java

```
package runner;
import io.cucumber.junit.Cucumber;
import io.cucumber.junit. CucumberOptions;
import org.junit.runner. RunWith;


@RunWith(Cucumber.class)
@CucumberOptions(

// first we need to teach our framework where all the features will be
located
features="src/test/resources/features",
// then we have to connect the feature steps with step definitions
// for this we will use "glue" to provide the location to step
definitions
// to provide location of steps we start wil by default "src/test/java"
glue="step_definitions"
)
public class TestRunner {}

// no code need inside of this class!
```

12. **Under utilities, Create DriverFactory java class to call the web browsers.**

```
package utilities;

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.edge.EdgeDriver;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.safari.SafariDriver;

public class DriverFactory {
    /**
     * The method takes a string as @param
     * It will decide what type of driver to create
```

```
     *
     * @return WebDriver
     */
    public static WebDriver getDriver(String browserType) {
        switch (browserType.toUpperCase()) {
            case "FIREFOX":
                return new FirefoxDriver();
            case "SAFARI":
                return new SafariDriver();
            case "EDGE":
                return new EdgeDriver();
            default:
                return new ChromeDriver();
        }
    }
```

13. **Under pages, create the LoginPage java class to login to the web page. This code represents these steps from user-login.feature file.**

```
    Feature: User Login Process

    Scenario: Login with valid credentials
        Given user navigates to hrm login page
        When user enters username and password
        And user clicks login button
```

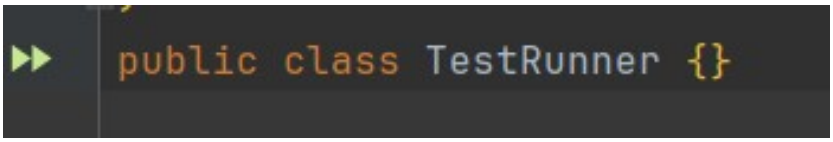
```
package pages;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.FindBy;
import org.openqa.selenium.support.PageFactory;
/**
 * LoginPage represent real HRM Login Page as a WebElements repository
 * The properties are public so we can access them directly
 * Here PageFactory is used to initialize the WebElements with @FindBy
annotations **/
public class LoginPage {

    public LoginPage(WebDriver driver) {

        PageFactory.initElements(driver, this);
    }
    @FindBy(id = "txtUsername")
    public WebElement usernameInputBox;
    @FindBy(id = "txtPassword")
    public WebElement passwordInputBox;
    @FindBy(id = "btnLogin")
```

```
    public WebElement loginButton;
    @FindBy(id = "spanMessage")
    public WebElement errorMessage;
}
```

14. **Under pages, create the HomePage java class to access to the homw page. This code represents the following step from user-login.feature file:**

```
    Then user is redirected to home page
```

15. **Run the TestRunner Class:**



dryRun = true

In Cucumber, a dry run refers to running the tests without actually executing the steps. This is useful to check that all the **step definitions** in the feature files have corresponding implementations in the step definition files.

To perform a dry run in Cucumber, you can use the `--dry-run` option. When you use this option, Cucumber will check the mappings between your feature files and step definitions but won't execute the test steps. This helps to identify any missing or incorrect step definitions before running the full tests.

dryRun = false

- **`dryRun = false`**: This is the default behavior. It means that Cucumber will run the tests by executing the step definitions linked to your feature files. The steps will actually be performed, interacting with the system (e.g., navigating a web browser, entering data, etc.).

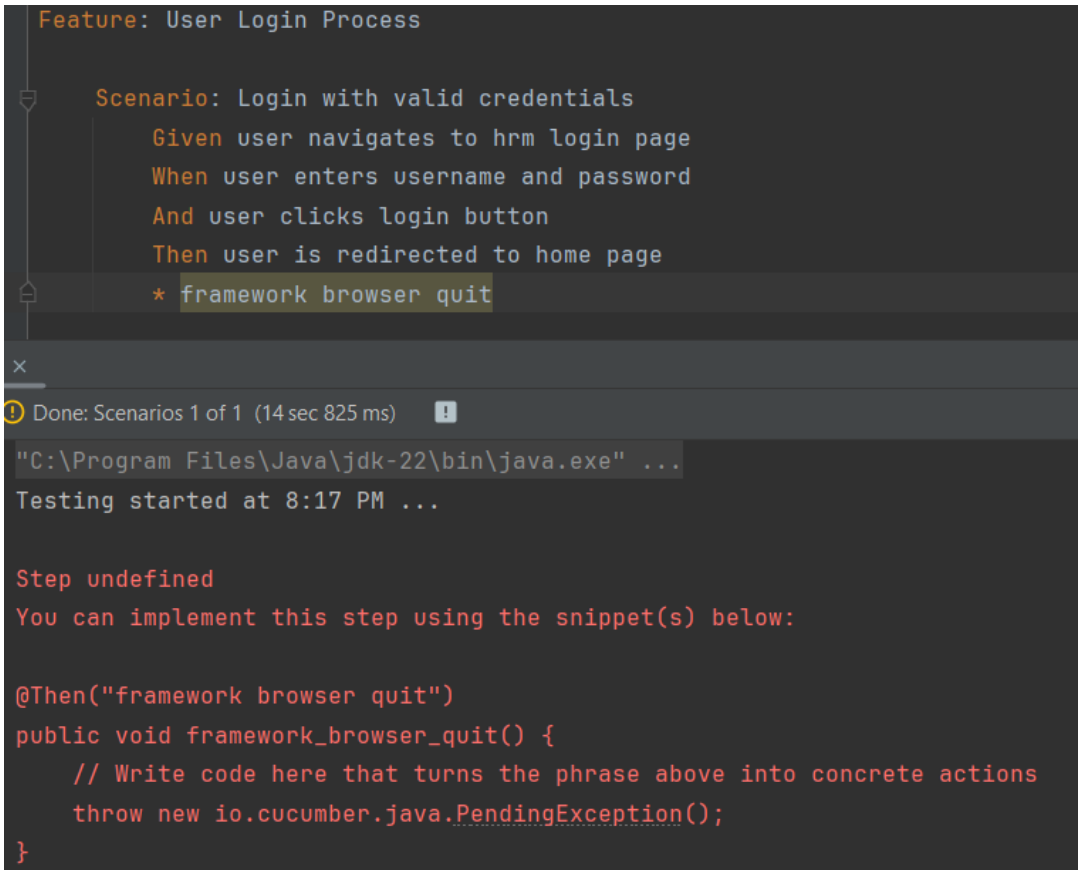
So, when you see `dryRun = false`, it implies that the test execution is **fully functional**, and the tests will be run for real.

* **framework browser quit**

The line * framework browser quit seems to be a shorthand or syntax used in some testing frameworks (possibly Karate or other BDD tools) to close or quit the browser at the end of a test scenario.
In general:

- framework likely refers to the automation framework being used.
- browser quit would tell the framework to quit the browser, effectively closing the browser window.

If this is part of a Karate DSL (a popular BDD testing framework similar to Cucumber), it typically follows this kind of syntax. In Karate, the * indicates a step or action, and browser quit would be the command to stop the browser after completing the scenario.
In summary, the line * framework browser quit means that after the test scenario finishes running, the browser instance is terminated or closed to ensure no browser windows are left open.


Feature: User Login Process
Scenario: Login with valid credentials
Given user navigates to hrm login page
When user enters username and password
And user clicks login button
Then user is redirected to home page
* framework browser quit
Done: Scenarios 1 of 1 (14 sec 825 ms)
"C:\Program Files\Java\jdk-22\bin\java.exe" ...
Testing started at 8:17 PM ...
Step undefined
You can implement this step using the snippet(s) below:
@Then("framework browser quit")
public void framework_browser_quit() {
// Write code here that turns the phrase above into concrete actions
throw new io.cucumber.java.PendingException();
}

16. To Add Another Scenario : you have to change dry run to true:

incomplete **dryRun = true**

17. Add tags:

```
tags= "@loginInvalid"
        //tags="@smoke"
        //tags="@loginInvalid and @smoke"
        //tags="@loginInvalid and @regression"
        //tags="@loginInvalid and @loginValid"
```

- Just like **TestNG was providing Groups** to run specific test cases together, Cucumber gives that option using **Tags**.
- We can tag scenarios like **@Smoke, @Regression**, @CustomName
  - `@Regression @Smoke`
  - `Scenario: Add item to cart`
  - `  Given I am on homepage`
- We can add multiple tags to one scenario
- We can run scenarios that have any of the mentioned tags (**OR** logic):
  - `tags = "@search or @smoke"`
- We can also run scenarios that have all the mentioned tags. (**AND** logic):
  - `tags = "@search and @smoke"`
  - Difference: each tag is in its own “” quotes.
- We can also ignore running certain tags
  - `tags = "@regression not @wip"`

In **Cucumber**, the `tags` option is used to filter and run specific scenarios or feature files based on tags that you assign to them. Tags allow you to categorize your tests and selectively execute scenarios that are relevant to a particular context, like smoke testing, regression testing, or testing specific functionality (e.g., invalid login).

Here’s a breakdown of how the tag syntax works with your example:

1. `tags = "@loginInvalid"`

- **Explanation**: This will run only the scenarios or feature files that are tagged with `@loginInvalid`.

  Example:

  ```
  @loginInvalid
  Scenario: Login with invalid credentials
    Given user navigates to hrm login page
    When user enters incorrect username and password
    Then user sees an error message
  ```

  The test runner will only run this scenario because it has the `@loginInvalid` tag.

2. `//tags = "@smoke"`

- **Explanation**: This is commented out, but if used, it would run all scenarios or features tagged with `@smoke`. This might be used for **smoke tests**, which are a subset of tests that check basic functionality.

  Example:

  ```
  @smoke
  Scenario: User login functionality
    Given user navigates to hrm login page
    When user enters valid credentials
    Then user is redirected to the home page
  ```

3. **`//tags = "@loginInvalid and @smoke"`**

- **Explanation**: This is commented out, but if used, it would run only the scenarios that are tagged with both `@loginInvalid` **and** `@smoke`. A scenario must have both tags to be executed.

  Example:

  ```
  @loginInvalid @smoke
  Scenario: Invalid login attempt for smoke test
    Given user navigates to hrm login page
    When user enters incorrect username and password
    Then user sees an error message
  ```

  This scenario will be executed because it has both `@loginInvalid` and `@smoke` tags.

4. **`//tags = "@loginInvalid and @regression"`**

- **Explanation**: This is commented out, but if used, it would run scenarios that are tagged with both `@loginInvalid` and `@regression`. These scenarios would typically be part of **regression testing**, which involves re-running functional tests to ensure that existing functionality still works after changes.

  Example:

  ```
  @loginInvalid @regression
  Scenario: Invalid login for regression testing
    Given user navigates to hrm login page
    When user enters an invalid username and password
    Then user sees an error message
  ```

5. **`//tags = "@loginInvalid and @loginValid"`**

- **Explanation**: This would run scenarios that have both `@loginInvalid` and `@loginValid` tags. However, in practice, it's unlikely to find scenarios tagged with both, since **valid** and **invalid** logins are generally two different cases. But you could use such a combination if a scenario tests both aspects.

  Example:

  ```
  @loginInvalid @loginValid
  ```

```
Scenario: Verify both valid and invalid login cases
  Given user navigates to hrm login page
  When user enters invalid credentials
  Then user sees an error message
  When user enters valid credentials
  Then user is redirected to the home page
```

Summary of Tag Usage:

- **Single Tag** (`@loginInvalid`): Run only scenarios with this tag.
- **Multiple Tags (AND condition)** (`@loginInvalid and @smoke`): Run scenarios that have **both** tags.

monochrome = true

In **Cucumber**, the monochrome = true setting is used to make the console output more readable by removing unnecessary characters and formatting, such as ANSI color codes, which can make the output look cluttered, especially in certain environments (like CI/CD pipelines or plain terminals).
Here’s what it means:

- **monochrome = true**: It ensures that the output is in plain text without any extra formatting or color, making it cleaner and easier to read.
- **monochrome = false** (default): The output includes additional formatting like colored text to make it more visually distinct, but it can add unnecessary characters in some environments.

In summary, setting monochrome = true makes the console output simpler and easier to read, especially if you're dealing with environments where color codes don't render well.

Cucumber Options with `monochrome = true`:

java

Copy code

```java
import io.cucumber.junit.Cucumber;
import io.cucumber.junit.CucumberOptions;
import org.junit.runner.RunWith;

@RunWith(Cucumber.class)
@CucumberOptions(
        features = "src/test/resources/features",
        glue = "step_definitions",
        monochrome = true  // This makes the console output cleaner
)
public class TestRunner {
}
```

Output when `monochrome = true`:

sql

Copy code

```sql
Feature: User Login Process

  Scenario: Login with valid credentials
    Given user navigates to hrm login page
    When user enters username and password
    And user clicks login button
    Then user is redirected to home page

1 Scenarios (1 passed)
4 Steps (4 passed)
0m3.657s
```

Output when `monochrome = false` (default):

```sql
Feature: User Login Process

[32mScenario: Login with valid credentials  [0m
[32mGiven[0m user navigates to hrm login page
[32mWhen[0m user enters username and password
[32mAnd[0m user clicks login button
[32mThen[0m user is redirected to home page

1 Scenarios ([32m1 passed[0m)
4 Steps ([32m4 passed[0m)
0m3.657s
```

In the second example, with `monochrome = false`, you see color codes (`[32m`, `[0m`) in the output, which can make it harder to read in certain environments. When `monochrome = true` is enabled, the output is clean and free of those codes, making it more readable.

Example of cucumber project with explanation

Let's walk through a **real-world example of a Cucumber project** step-by-step, covering both the structure and the explanation behind each component. This will help clarify how Cucumber works, how it integrates with Selenium WebDriver, and how you can use it to build test automation for web applications.

Scenario: Testing the Login Functionality of an E-commerce Website

We'll be testing a simple login functionality for an e-commerce website. The project will use **Cucumber**, **Selenium WebDriver**, and **JUnit** to automate the tests. For simplicity, the login page will consist of a username field, password field, and login button.

Project Structure

Here’s what the basic structure of your Cucumber project looks like:

```
src
└── test
```

```
    └── java
        └── step_definitions
            └── LoginSteps.java
        └── runners
            └── TestRunner.java
        └── pages
            └── LoginPage.java
        └── features
            └── Login.feature
        └── utilities
            └── DriverFactory.java
```

1. Feature File (`Login.feature`)

The **feature file** defines the high-level behavior in Gherkin syntax, which is easy to understand by both technical and non-technical people. This file specifies what should happen in plain language.

`Login.feature`

```
Feature: Login Feature

  Scenario: Successful login with valid credentials
    Given the user navigates to the login page
    When the user enters a valid username and password
    And clicks the login button
    Then the user should be redirected to the dashboard
```

Explanation:

- `Feature`: This keyword represents the functionality under test (Login feature).
- `Scenario`: This describes the specific test case for the feature (login with valid credentials).
- `Given`, `When`, `Then`: These are Gherkin keywords used to define the steps in the scenario.
    - `Given`: Precondition before performing the action.
    - `When`: The action being performed.
    - `Then`: The expected result.

---

2. Step Definitions (`LoginSteps.java`)

The **step definition file** maps the Gherkin steps from the feature file to Java methods using Selenium WebDriver to interact with the web application.

`LoginSteps.java`

```
package step_definitions;

import io.cucumber.java.en.Given;
import io.cucumber.java.en.Then;
import io.cucumber.java.en.When;
import org.junit.Assert;
import org.openqa.selenium.WebDriver;
```

```
import pages.LoginPage;
import utilities.DriverFactory;

public class LoginSteps {

    private WebDriver driver = DriverFactory.getDriver("CHROME");
    private LoginPage loginPage = new LoginPage(driver);

    @Given("the user navigates to the login page")
    public void the_user_navigates_to_the_login_page() {
        driver.get("https://example.com/login");
    }

    @When("the user enters a valid username and password")
    public void the_user_enters_a_valid_username_and_password() {
        loginPage.enterUsername("validUser");
        loginPage.enterPassword("validPass");
    }

    @When("clicks the login button")
    public void clicks_the_login_button() {
        loginPage.clickLoginButton();
    }

    @Then("the user should be redirected to the dashboard")
    public void the_user_should_be_redirected_to_the_dashboard() {
        Assert.assertTrue(loginPage.isUserOnDashboard());
    }
}
```

Explanation:

- `@Given`, `@When`, `@Then`: These annotations map the Gherkin steps from the feature file to corresponding Java methods.
- The WebDriver instance is initialized using the `DriverFactory`.
- The `LoginPage` class contains the logic for interacting with elements on the login page (discussed next).
- Inside each step method, the page object methods are invoked to perform actions like entering credentials and clicking the login button.
- Assertions (JUnit's `Assert`) are used to verify the outcome of the test.

---

3. Page Object (`LoginPage.java`)

The **Page Object Model (POM)** is a design pattern used to create an object repository for web elements. Each web page in the application has a corresponding page object class to interact with elements.

`LoginPage.java`

```
package pages;

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.FindBy;
import org.openqa.selenium.support.PageFactory;
```

```
public class LoginPage {

    WebDriver driver;

    // Constructor to initialize the driver
    public LoginPage(WebDriver driver) {
        this.driver = driver;
        PageFactory.initElements(driver, this);
    }

    // Web elements on the login page
    @FindBy(id = "username")
    private WebElement usernameInput;

    @FindBy(id = "password")
    private WebElement passwordInput;

    @FindBy(id = "loginButton")
    private WebElement loginButton;

    // Actions on the login page
    public void enterUsername(String username) {
        usernameInput.sendKeys(username);
    }

    public void enterPassword(String password) {
        passwordInput.sendKeys(password);
    }

    public void clickLoginButton() {
        loginButton.click();
    }

    public boolean isUserOnDashboard() {
        // Simplified check for example purposes (could check a dashboard-specific
element)
        return driver.getTitle().contains("Dashboard");
    }
}
```

Explanation:

- The **page object** class contains web elements and methods to interact with them.
- `@FindBy`: Used to locate the elements on the page.
- The actions (`enterUsername`, `enterPassword`, etc.) abstract the operations on web elements, allowing for more maintainable and reusable code.

---

4. Driver Factory (`DriverFactory.java`)

The **DriverFactory** is a utility class to initialize the WebDriver based on the browser specified.

`DriverFactory.java`

```
package utilities;

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;

public class DriverFactory {

    private static WebDriver driver;

    public static WebDriver getDriver(String browser) {
        if (driver == null) {
            if (browser.equalsIgnoreCase("CHROME")) {
                System.setProperty("webdriver.chrome.driver", "path/to/chromedriver");
                driver = new ChromeDriver();
            }
            // Additional browser setups (e.g., Firefox, Edge) can be added here
        }
        return driver;
    }

    public static void closeDriver() {
        if (driver != null) {
            driver.quit();
            driver = null;
        }
    }
}
```

Explanation:

- The `getDriver` method initializes and returns a WebDriver instance based on the provided browser type (in this case, Chrome).
- `closeDriver` is used to close the browser session after the test is completed.

---

5. Test Runner (`TestRunner.java`)

The **TestRunner** class uses Cucumber options to specify where the feature files and step definitions are located and runs the tests with JUnit.

`TestRunner.java`

```
package runners;

import io.cucumber.junit.Cucumber;
import io.cucumber.junit.CucumberOptions;
import org.junit.runner.RunWith;

@RunWith(Cucumber.class)
@CucumberOptions(
    features = "src/test/resources/features", // Path to feature files
    glue = "step_definitions",                // Path to step definitions
    plugin = {"pretty", "html:target/cucumber-reports"}, // Report generation
    monochrome = true                          // Clean console output
)
public class TestRunner {
}
```

Explanation:

- `@RunWith(Cucumber.class)`: Specifies that the class should be run as a Cucumber test.
- `@CucumberOptions`: Configures Cucumber to run feature files located in the specified path and step definitions in the specified package.
- `plugin`: Specifies the report format (`pretty` for readable output, `html` for HTML reports).

---

6. Running the Tests

To run the Cucumber tests:

1. Open your IDE (e.g., IntelliJ or Eclipse).
2. Right-click on the `TestRunner.java` and select **Run**.
3. Alternatively, you can run the tests through the terminal:

```
bash
Copy code
mvn test
```

7. pom.xml (For Maven Projects)

Here are the dependencies you'll need to add to your **`pom.xml`** file if you're using Maven:

```
<dependencies>
    <!-- Cucumber dependencies -->
    <dependency>
        <groupId>io.cucumber</groupId>
        <artifactId>cucumber-java</artifactId>
        <version>7.0.0</version>
    </dependency>
    <dependency>
        <groupId>io.cucumber</groupId>
        <artifactId>cucumber-junit</artifactId>
        <version>7.0.0</version>
    </dependency>

    <!-- Selenium dependencies -->
    <dependency>
        <groupId>org.seleniumhq.selenium</groupId>
        <artifactId>selenium-java</artifactId>
        <version>4.0.0</version>
    </dependency>

    <!-- JUnit for assertions -->
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.13.2</version>
    </dependency>
</dependencies>
```

Summary of the Workflow:

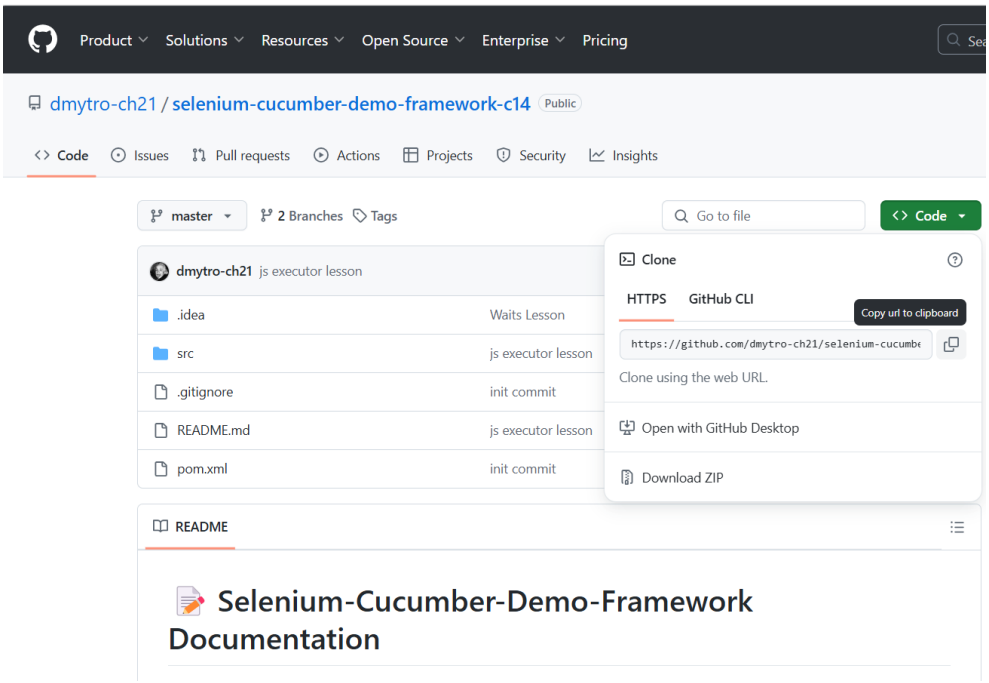
1. **Feature File** (`Login.feature`): Contains human-readable test scenarios written in Gherkin.
2. **Step Definitions** (`LoginSteps.java`): Maps each Gher

Copying Documentation to README.md File :

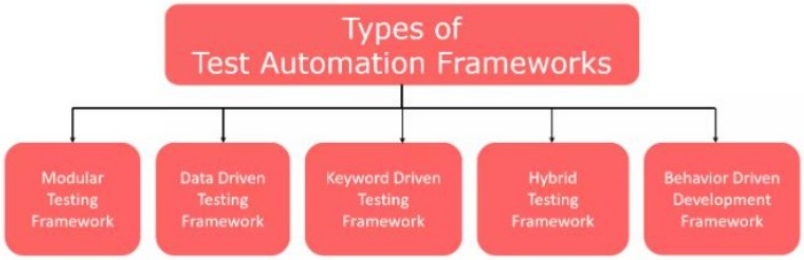
Copy url that contains files from **Code** to IntelliJ

GitHub - dmytro-ch21/selenium-cucumber-demo-framework-c14

Go to README.md, copy raw



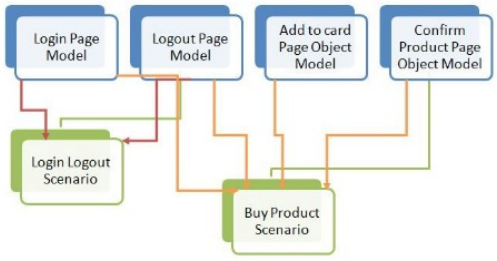
Testing Frameworks



- These frameworks are user-defined
- They are different than predefined frameworks such as:
  - JUnit, TestNG, Cucumber etc.

Page Object Model

- **Page Object Model** → is a design pattern that includes **Object Repositories** for web-elements.
  - web pages are represented as java classes.
  - *Login page, Home page, About Us page, Search Results page etc.*
- All the web-elements of each page are located and stored on corresponding PO class.
  - *loginButton, username inputbox, password input box etc.*
- All reusable methods of that web-page are stored in libraries.
  - login(String username, String password);
- Advantages:
  - Test cases are separated from UI web elements and actions.
  - **Reusability** of code
  - Easy **Maintenance**



A modular testing framework is a type of software testing framework that emphasizes creating small, independent, reusable modules or functions to test specific parts of an application. Each module represents a distinct part of the functionality, and tests are designed to validate these modules individually. By focusing on small, isolated components, modular testing frameworks support maintainable, flexible, and scalable test automation.

Key Characteristics of a Modular Testing Framework:

1. **Modularity**: The application is divided into smaller, isolated modules, and each module has a corresponding test function.
2. **Reusability**: Each test module is reusable, so it can be combined with others to form larger, comprehensive test suites.
3. **Maintainability**: Because tests are divided into modules, changes to one part of the application only require updating a specific test module.
4. **Scalability**: As new features are added, new test modules can be created without impacting existing tests.
5. **Separation of Concerns**: Each module is responsible for testing a particular feature, function, or component, which promotes cleaner and more organized test code.

Benefits of Modular Testing Frameworks:

- **Reduces Redundancy**: With reusable modules, the same test steps do not need to be repeated across multiple test cases.
- **Improves Maintenance**: Changes to the application under test (AUT) can often be managed by updating individual modules instead of the entire test suite.
- **Enhanced Collaboration**: Modular design allows multiple team members to work on different modules simultaneously.
- **Efficient Debugging**: When a test fails, it’s easier to pinpoint the issue because each module covers a distinct function.

Structure of a Modular Testing Framework:

In a modular testing framework, the structure is often organized in a way that each module corresponds to a specific feature or functionality. A basic structure might look like this:

plaintext

Copy code

```
project-folder/
│
├── tests/
│   ├── login/
│   │   ├── loginTest.java
│   │   ├── invalidLoginTest.java
│   │   └── logoutTest.java
│   ├── shoppingCart/
```

```
│   │   ├── addItemTest.java
│   │   ├── removeItemTest.java
│   │   └── checkoutTest.java
│   └── userProfile/
│       ├── updateProfileTest.java
│       └── viewProfileTest.java
│
├── modules/
│   ├── LoginModule.java
│   ├── ShoppingCartModule.java
│   └── UserProfileModule.java
│
├── utils/
│   ├── DriverFactory.java
│   ├── ConfigReader.java
│   └── Reporting.java
│
└── testng.xml
```

How Modular Testing Works:

1. **Creating Modules**: Each major functionality, like login, shopping cart, or user profile, is broken down into modules. Each module has its own methods for interacting with and validating the application's behavior for that feature.
2. **Building Test Cases**: Test cases are built by combining various modules. For example, a checkout test might use the login module, shopping cart module, and payment module.
3. **Reusable Components**: Common actions (like login or logout) are created as reusable components and can be called multiple times in different test scenarios.
4. **Driver and Utility Layer**: Utilities like DriverFactory for managing browser sessions, ConfigReader for reading configurations, and reporting tools are often included to support the modular tests.

Example of a Modular Testing Framework (using Java & TestNG):

Consider a web application with the following functionalities: **Login**, **Shopping Cart**, and **User Profile**.

Module Example - LoginModule.java:

This module might contain all login-related methods and interactions.

```
package modules;


import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.FindBy;


public class LoginModule {


    WebDriver driver;


    @FindBy(id = "username")
    WebElement usernameInput;


    @FindBy(id = "password")
    WebElement passwordInput;


    @FindBy(id = "loginButton")
    WebElement loginButton;


    public LoginModule(WebDriver driver) {
        this.driver = driver;
    }


    public void login(String username, String password) {
        usernameInput.sendKeys(username);
        passwordInput.sendKeys(password);
```

```
        loginButton.click();
    }
}
```

Test Case Example - LoginTest.java:

The test case imports the LoginModule and uses its reusable login function to run tests.

```
package tests.login;

import org.testng.Assert;
import org.testng.annotations.Test;
import modules.LoginModule;
import utilities.DriverFactory;

public class LoginTest {

    @Test
    public void testValidLogin() {
        LoginModule loginModule = new LoginModule(DriverFactory.getDriver());
        loginModule.login("validUser", "validPassword");
        // Assert expected outcomes
        Assert.assertTrue(/*condition for successful login*/);
    }
}
```

Key Components of a Modular Testing Framework:

1. **Modules**: Individual components for different features (like Login, Profile, Cart) that contain reusable methods for those features.
2. **Test Suites**: Organize multiple test cases, usually using a test runner like **JUnit** or **TestNG**.
3. **Driver Management**: Utility for managing browser drivers (e.g., **DriverFactory**).
4. **Configuration Management**: Store and read configurations from a centralized location (e.g., **ConfigReader**).

5. **Logging and Reporting**: Logging utilities and reporting frameworks to summarize test results.

Popular Tools and Frameworks for Modular Testing:

1. **JUnit / TestNG**: These test runners can be used to execute modular tests in Java.
2. **Selenium WebDriver**: For automating browser interactions.
3. **Page Object Model (POM)**: Although POM is a design pattern, it's often used with modular frameworks to keep UI elements separate from test logic, making tests more modular.
4. **Cucumber**: When combined with BDD, Cucumber can also support modular test writing.
5. **Robot Framework**: This framework is inherently modular and is organized around reusable keywords that represent individual test actions.

Best Practices for Modular Testing:

- **Isolate Modules**: Ensure that each module is independent and only focused on one function or component.
- **Encapsulate Logic**: Keep business logic in modules and make them reusable so that test cases remain concise.
- **Avoid Hardcoding**: Use configurations or data-driven approaches to keep your modules adaptable.
- **Organize by Feature**: Group modules and tests by feature or functionality to improve code readability.
- **Combine with POM**: Implement Page Object Model for better separation between test logic and UI structure.

When to Use Modular Testing Frameworks:

Modular testing frameworks are especially beneficial for:

- **Large applications** with multiple, independent functionalities.
- **Scalable projects** where new features are added regularly.
- **Collaborative environments** where multiple team members develop and maintain different parts of the test suite.
- **Projects with frequent changes** that require isolated updates in specific modules without breaking the overall test suite.

Conclusion:

A modular testing framework provides flexibility, scalability, and maintainability by breaking down tests into independent modules. This approach ensures that tests are

manageable and maintainable, making it easier to add new tests or modify existing ones. By focusing on modularity and reusability, this framework aligns well with Agile and CI/CD environments, supporting continuous development and testing.

A data-driven testing framework is a type of automated testing framework that focuses on running tests using different sets of input data. Instead of hardcoding the values in the test scripts, the framework pulls input values from external data sources, such as spreadsheets, databases, or XML/JSON files. This allows the same test script to be executed multiple times with various data sets, significantly reducing the amount of code while increasing test coverage.

Key Characteristics of a Data-Driven Testing Framework:

1. **Separation of Data and Logic**: Test data is separated from test scripts, which keeps tests readable and maintainable.
2. **Reusability**: The same test can be reused with different data sets, making it efficient for testing multiple scenarios.
3. **Flexibility and Scalability**: Easy to add more test cases simply by adding more data, without needing to write additional scripts.
4. **Centralized Data Management**: Test data is managed in one location, which simplifies updates and maintenance.

Structure of a Data-Driven Testing Framework:

A typical structure for a data-driven testing framework looks like this:

plaintext

Copy code

```
project-folder/
│
├── tests/
│   ├── loginTest.java
│   ├── searchTest.java
│   └── checkoutTest.java
│
├── data/
│   ├── loginData.csv
│   ├── searchData.xlsx
│   └── checkoutData.json
```

```
|
├── utilities/
|   ├── DataProviderUtility.java
|   ├── ExcelReader.java
|   └── JSONReader.java
|
└── testng.xml
```

Benefits of Data-Driven Testing:

- **Enhanced Test Coverage**: Easily run the same test with different data sets, enabling broader coverage of scenarios.
- **Easy Maintenance**: Any change in the data (e.g., usernames, passwords, expected results) can be managed centrally without altering test scripts.
- **Reduced Code Duplication**: Data-driven tests can be re-executed with new data, avoiding redundant code for similar tests.

Implementing a Data-Driven Framework:

Here's how to implement a data-driven testing framework in a typical environment using **Java** and **TestNG**:

Step 1: Set Up the Data Source

Data can be stored in various formats, such as **CSV**, **Excel**, **XML**, or **JSON**. In this example, let's use an **Excel** file.

For instance, an Excel file loginData.xlsx might look like this:

Username	Password	ExpectedMessage
user1	pass123	Login Successful
user2	wrongPass	Invalid Password
invalidUser	pass123	Invalid Username

Step 2: Create a Data Provider to Fetch Data

In TestNG, you can use the @DataProvider annotation to supply data to test methods.

Here's an example of DataProviderUtility.java:

```
import org.testng.annotations.DataProvider;
```

```
import utilities.ExcelReader;


public class DataProviderUtility {


    @DataProvider(name = "loginData")
    public static Object[][] getLoginData() {
        return ExcelReader.getExcelData("data/loginData.xlsx", "Sheet1");
    }
}
```

Step 3: Use Data in Test Cases

In your test script, annotate the test method with @Test(dataProvider = "dataProviderName") and include the parameters for each data column.

Example loginTest.java:

```
import org.testng.Assert;
import org.testng.annotations.Test;
import pages.LoginPage;
import utilities.DriverFactory;


public class LoginTest {


    @Test(dataProvider = "loginData", dataProviderClass = DataProviderUtility.class)
    public void testLogin(String username, String password, String expectedMessage) {
        LoginPage loginPage = new LoginPage(DriverFactory.getDriver());
        loginPage.login(username, password);
        String actualMessage = loginPage.getLoginMessage();
        Assert.assertEquals(actualMessage, expectedMessage, "Login message
mismatch!");
    }
}
```

Step 4: Implement Data Extraction Utility

This utility reads the data from the external data source, such as an Excel file. Here’s an example for reading data from Excel (ExcelReader.java):

```
import org.apache.poi.ss.usermodel.Row;

import org.apache.poi.ss.usermodel.Sheet;

import org.apache.poi.ss.usermodel.Workbook;

import org.apache.poi.xssf.usermodel.XSSFWorkbook;


import java.io.FileInputStream;

import java.io.IOException;

import java.util.ArrayList;

import java.util.Iterator;

import java.util.List;


public class ExcelReader {


    public static Object[][] getExcelData(String filePath, String sheetName) {

        List<Object[]> data = new ArrayList<>();

        try (FileInputStream fis = new FileInputStream(filePath);

             Workbook workbook = new XSSFWorkbook(fis)) {


            Sheet sheet = workbook.getSheet(sheetName);

            Iterator<Row> rowIterator = sheet.iterator();


            // Skip header row

            rowIterator.next();


            while (rowIterator.hasNext()) {

                Row row = rowIterator.next();

                String username = row.getCell(0).getStringCellValue();

                String password = row.getCell(1).getStringCellValue();
```

```
            String expectedMessage = row.getCell(2).getStringCellValue();
            data.add(new Object[]{username, password, expectedMessage});
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
    return data.toArray(new Object[0][]);
  }
}
```

Running the Data-Driven Test:

To run the test, specify it in the TestNG configuration file or execute it from the command line/IDE. Each row in the Excel file will be a new test execution, providing different data for each run.

Example of Data-Driven Framework Components:

1. **Data Providers**: Supply data for tests, often leveraging tools like Apache POI for Excel, Jackson for JSON, or OpenCSV for CSV.
2. **Test Data Management**: Organize data files in a dedicated folder and ensure data is meaningful and covers various edge cases.
3. **Test Execution**: Typically, a test runner (like TestNG or JUnit) reads data through the @DataProvider or similar mechanisms to execute multiple tests.
4. **Assertions and Validations**: Each test run will validate outputs against expected results, allowing multiple data-driven assertions.

Popular Data-Driven Testing Tools:

- **JUnit and TestNG** (Java): Support data providers natively.
- **NUnit** (.NET): Provides a TestCaseSource attribute for data-driven tests.
- **Selenium**: Combined with other frameworks (e.g., TestNG or JUnit) for web testing.
- **Robot Framework**: Uses data-driven testing keywords and syntax for table-based data.
- **Cucumber**: Supports scenario outlines and example tables for data-driven tests in Gherkin syntax.

Advantages of Data-Driven Testing Frameworks:

- **Efficient Coverage**: Test multiple input variations with one script.

- **Easier Maintenance**: Separate data from test logic, allowing test logic changes without affecting data.
- **Reduced Test Code**: Reuse the same test for different data sets, reducing the amount of test code.
- **Scalability**: Easily expand testing by adding new data without creating new test scripts.

Conclusion:

Data-driven testing frameworks are an effective solution for automating test cases that need to be run with various data sets. They improve test efficiency and coverage, especially when working with applications that require testing different input combinations. By separating data from test scripts, a data-driven testing framework makes test maintenance easier and supports scalability in growing applications.

A Behavior-Driven Development (BDD) framework is a testing approach that emphasizes collaboration between developers, testers, and business stakeholders to define test cases in a natural language that everyone can understand. BDD frameworks are based on the premise of writing tests in a way that reflects user behavior and expectations, often written in Given-When-Then format to describe scenarios. This approach helps ensure that the software meets the requirements by focusing on user actions and outcomes rather than on the technical details.

Key Concepts of BDD:

1. **Collaboration**: Encourages team members from different roles to work together, ensuring tests align with business requirements.
2. **Natural Language Syntax**: Test scenarios are written in plain English, making them understandable to non-technical stakeholders.
3. **Specification by Example**: Requirements are defined as examples or scenarios, providing clarity on expected behaviors.
4. **Executable Specifications**: Test scenarios act as both documentation and executable code, validating the application's behavior.

Structure of a BDD Framework:

A typical structure of a BDD framework involves organizing test cases and feature files in a way that is easy to read and follow:

```
project-folder/
│
├── features/                       # Holds all feature files
│   ├── login.feature
│   └── search.feature
│
├── step_definitions/               # Contains step definitions for each feature
│   ├── LoginSteps.java
│   └── SearchSteps.java
│
├── runners/                        # Test runners for organizing and running tests
```

```
│   └── TestRunner.java
│
└── utils/                          # Utilities and helper classes
    ├── DriverFactory.java
    └── ConfigReader.java
```

Components of a BDD Framework:

1. **Feature Files**: Written in plain English, often using the Gherkin language. Each feature file describes a user story, with multiple scenarios for different behaviors.
2. **Step Definitions**: These are methods that map to each step in the feature files, containing the code to execute those steps.
3. **Test Runner**: Executes all or specific tests, often using frameworks like JUnit or TestNG.
4. **Utilities**: Supporting classes for common actions, configuration, or driver management.

Example of a BDD Scenario Using Cucumber (in Gherkin syntax):

Imagine a simple scenario for testing a login functionality:

`login.feature`:

```
Feature: Login functionality

  Scenario: Valid user login
    Given the user is on the login page
    When the user enters a valid username and password
    And clicks the login button
    Then the user should be redirected to the home page

  Scenario: Invalid login attempt
    Given the user is on the login page
    When the user enters an invalid username or password
    And clicks the login button
    Then an error message should be displayed
```

Step Definitions in Java (`LoginSteps.java`):

The step definitions correspond to each line in the feature file and contain the code to perform these actions.

```
package step_definitions;

import io.cucumber.java.en.Given;
import io.cucumber.java.en.When;
import io.cucumber.java.en.Then;
import org.junit.Assert;
import pages.LoginPage;
import pages.HomePage;
import utilities.DriverFactory;

public class LoginSteps {
    private LoginPage loginPage = new LoginPage(DriverFactory.getDriver());
```

```
    @Given("the user is on the login page")
    public void user_is_on_the_login_page() {
        loginPage.navigateToLoginPage();
    }

    @When("the user enters a valid username and password")
    public void user_enters_valid_credentials() {
        loginPage.enterUsername("validUser");
        loginPage.enterPassword("validPassword");
    }

    @When("the user enters an invalid username or password")
    public void user_enters_invalid_credentials() {
        loginPage.enterUsername("invalidUser");
        loginPage.enterPassword("invalidPassword");
    }

    @When("clicks the login button")
    public void user_clicks_login_button() {
        loginPage.clickLogin();
    }

    @Then("the user should be redirected to the home page")
    public void user_is_redirected_to_home_page() {
        HomePage homePage = new HomePage(DriverFactory.getDriver());
        Assert.assertTrue(homePage.isWelcomeMessageDisplayed());
    }

    @Then("an error message should be displayed")
    public void error_message_is_displayed() {
        Assert.assertTrue(loginPage.isErrorMessageDisplayed());
    }
}
```

Advantages of Using BDD Frameworks:

- **Clear Documentation**: Since BDD tests are written in plain language, they serve as living documentation that anyone on the team can understand.
- **Early Defect Detection**: Collaboration between team members ensures that requirements are correctly captured and understood before implementation.
- **Focused Development**: Developers have a clearer understanding of how the feature should behave from a user perspective.
- **Enhanced Test Coverage**: Scenarios cover various edge cases and workflows that reflect real user behavior.

Common Tools for BDD Frameworks:

- **Cucumber**: Widely used with Gherkin syntax for writing feature files and supports Java, Python, Ruby, etc.
- **JBehave**: A BDD framework for Java, similar to Cucumber but with a slightly different syntax and structure.
- **SpecFlow**: The BDD framework for .NET, allowing tests to be written in Gherkin.
- **Behave**: A BDD tool for Python that uses Gherkin and integrates well with Python-based projects.

Key Components and Workflow in a BDD Framework:

1. **Define User Stories**: Identify the main features or user stories to be tested and document them as feature files.
2. **Write Scenarios**: Describe how the system should behave in different scenarios for each feature.
3. **Write Step Definitions**: Implement methods that map to the Gherkin steps, executing the actual test code.
4. **Run Tests**: Use a test runner like JUnit or TestNG to execute the feature files. Each scenario in the feature file is treated as a separate test case.
5. **Review and Refine**: Stakeholders review the test results, and if issues are identified, the tests or implementation can be refined.

Example of a Cucumber Test Runner (`TestRunner.java`):

This runner class specifies where to find the feature files and step definitions and can be configured for reporting.

```
package runners;

import org.junit.runner.RunWith;
import io.cucumber.junit.Cucumber;
import io.cucumber.junit.CucumberOptions;

@RunWith(Cucumber.class)
@CucumberOptions(
        features = "src/test/resources/features",
        glue = "step_definitions",
        plugin = {"pretty", "html:target/cucumber-reports"},
        monochrome = true
)
public class TestRunner {
}
```

Reporting in BDD Frameworks:

Most BDD frameworks support reporting, which provides a detailed summary of test execution. Common reporting tools for BDD frameworks include:

- **Cucumber Reports**: Generates HTML reports summarizing each scenario and step.
- **Extent Reports**: Integrates with various test frameworks to provide rich, interactive reports.
- **Allure**: Works well with BDD frameworks, offering insights into test execution with graphical representations.

Best Practices for BDD:

1. **Involve Stakeholders Early**: Make sure that requirements and scenarios are clearly understood and agreed upon.
2. **Keep Scenarios Short and Simple**: Avoid complex scenarios to keep tests readable and maintainable.

3. **Focus on Behavior, Not Implementation**: Describe the "what" (expected behavior) rather than the "how" (implementation details).
4. **Use Tags for Organizing Tests**: Organize scenarios with tags (e.g., `@smoke`, `@regression`) to enable selective test execution.
5. **Reuse Step Definitions**: Keep step definitions generic enough for reuse across multiple scenarios.

When to Use BDD:

BDD frameworks are ideal for projects that benefit from:

- **Close collaboration** between stakeholders, developers, and testers.
- **Complex business rules** where the requirements need to be clearly understood and documented.
- **Agile methodologies**, as BDD aligns well with iterative development.
- **Cross-functional teams** where non-technical members can contribute to test design and review.

Conclusion:

A BDD framework offers a structured way to develop software with a focus on user behavior. By providing a natural language format, it enables better communication and collaboration, ensures alignment with requirements, and facilitates early bug detection. BDD is especially effective for Agile projects, where clear communication and regular iterations are critical to success.

Acceptance Testing Frameworks:

- **Natural language-based scenarios**: Many acceptance testing frameworks allow the use of plain language (such as Gherkin) so that non-technical stakeholders can understand the tests.
- **End-to-end testing**: They test the entire workflow of an application, from input to output.
- **User-focused validation**: They ensure that the software meets the requirements of the users or clients.
- **Integration with other testing tools**: These frameworks often integrate with unit testing, performance testing, and functional testing frameworks.

Popular Acceptance Testing Frameworks:

1. Cucumber

- **Language**: Supports Java, JavaScript, Ruby, Python, and others.
- **Description**: Cucumber is one of the most widely-used acceptance testing frameworks that uses the **Gherkin** language for writing test cases in a human-readable format. It follows the **Behavior-Driven Development (BDD)**

approach, where tests are written in a way that everyone (including non-technical stakeholders) can understand.

- **Example**:

gherkin

Feature: Login Functionality

Scenario: Successful login
  Given user is on the login page
  When user enters valid username and password
  Then user is redirected to the home page

Feature: User Login Process
  @loginValid @smoke
  Scenario: Login with valid credentials
    Given user navigates to hrm login page
    When user enters username and password
    And user clicks login button
    Then user is redirected to home page
    * framework browser quit

  @loginInvalid @smoke
  Scenario: Login with invalid credentials
    Given user navigates to hrm login page
    When user enters invalid username and password
    And user clicks login button
    Then user can see an error message
    * framework browser quit
  @emptyUsername

```
    Scenario: Login with invalid credentials when username empty
        When user enters empty for username and invalid password
        And user clicks login button
        Then user can see an error message
        * framework browser quit

    @emptyPassword
    Scenario: Login with invalid credentials when password empty
        When user enters invalid username and empty for password
        And user clicks login button
        Then user can see an error message that says Password cannot be empty
        * framework browser quit
```

++++++++++++

```
package step_definitions;

import io.cucumber.java.en.Given;
import io.cucumber.java.en.Then;
import io.cucumber.java.en.When;
import org.junit.Assert;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.support.ui.ExpectedConditions;
import org.openqa.selenium.support.ui.WebDriverWait;
import utilities.DriverFactory;
import pages.HomePage;// import LoginPage class
import pages.LoginPage; // Import LoginPage class
public class LoginSteps {
```

```
    private WebDriver driver = DriverFactory.getDriver ("CHROME");

    private LoginPage loginPage = new LoginPage(driver);
    private HomePage homePage = new HomePage(driver);  // Instantiate HomePage
    //First Scenario:

    @Given("user navigates to hrm login page")
    public void user_navigates_to_hrm_login_page() {
        driver.get("http://dev-hrm.yoll.io/index.php/auth/login");
    }
    @When("user enters username and password")
    public void user_enters_username_and_password() {
        loginPage.usernameInputBox.sendKeys("yoll-student");
        loginPage.passwordInputBox.sendKeys( "Bootcamp5#");
    }
    @When("user clicks login button")
    public void user_clicks_login_button() {
        loginPage.loginButton.click();

    }
    @Then("user is redirected to home page")
    public void user_is_redirected_to_home_page() {
        String actualWelcomeMessage = homePage.welcomeMessage.getText();
        String expectedWelcomeMessage = "Welcome Yoll";
// we will need to assert actual and expected
// The testNG structure of assert: Assert.assertEquals (actual, expected, message)
// In JUnit there are identical hard assert methods
        //Assert must be JUnit
```

```
        Assert.assertEquals(  "Welcome message verification failed",
expectedWelcomeMessage, actualWelcomeMessage);


    }
    @Then("framework browser quit")
    public void framework_browser_quit() {
        driver.quit();
    }


    //Second Scenario:
    @When("user enters invalid username and password")
    public void user_enters_invalid_username_and_password() {
        loginPage.usernameInputBox.sendKeys("invalid");
        loginPage.passwordInputBox.sendKeys( "invalid");
    }
    @Then("user can see an error message")
    public void user_can_see_an_error_message() {
        String actualErrorMessage = loginPage.errorMessage.getText();
        String expectedErrorMessage = "Invalid credentials";
        Assert.assertEquals(  "Error message verification failed",
expectedErrorMessage,actualErrorMessage);
    }
    @When("user enters empty for username and invalid password")
    public void user_enters_empty_for_username_and_invalid_password() {


        loginPage.usernameInputBox.sendKeys("");
        loginPage.passwordInputBox.sendKeys( "invalid");
    }
    @Then("user can see an error message that says username cannot be empty")
    public void
user_can_see_an_error_message_that_says_username_cannot_be_empty() {
```

```
        String actualErrorMessage = loginPage.errorMessage.getText();
        String expectedErrorMessage = "Username cannot be empty";
        Assert.assertEquals(  "Error message verification failed",
expectedErrorMessage,actualErrorMessage);
    }
    @When("user enters invalid username and empty for password")
    public void user_enters_invalid_username_and_empty_for_password() {
        loginPage.usernameInputBox.sendKeys("invalid");
        loginPage.passwordInputBox.sendKeys( "");
    }
    @Then("user can see an error message that says Password cannot be empty")
    public void user_can_see_an_error_message_that_says_password_cannot_be_empty()
{
        String actualErrorMessage = loginPage.errorMessage.getText();
        String expectedErrorMessage = "Password cannot be empty";
        Assert.assertEquals(  "Error message verification failed",
expectedErrorMessage,actualErrorMessage);
    }


}
```

LoginSteps

```
package step_definitions;

import io.cucumber.java.en.Given;
import io.cucumber.java.en.Then;
import io.cucumber.java.en.When;
import org.junit.Assert;
import org.openqa.selenium.WebDriver;

import utilities.DriverFactory;
import pages.HomePage;
import pages.LoginPage;

public class LoginSteps {
```

```
    private WebDriver driver = DriverFactory.getDriver("CHROME");
    private LoginPage loginPage = new LoginPage(driver);
    private HomePage homePage = new HomePage(driver);

    // First Scenario:
    @Given("user navigates to hrm login page")
    public void user_navigates_to_hrm_login_page() {
        driver.get("http://dev-hrm.yoll.io/index.php/auth/login");
    }

    @When("user enters username and password")
    public void user_enters_username_and_password() {
        loginPage.usernameInputBox.sendKeys("yoll-student");
        loginPage.passwordInputBox.sendKeys("Bootcamp5#");
    }

    @When("user clicks login button")
    public void user_clicks_login_button() {
        loginPage.loginButton.click();
    }

    @Then("user is redirected to home page")
    public void user_is_redirected_to_home_page() {
        String actualWelcomeMessage = homePage.welcomeMessage.getText();
        String expectedWelcomeMessage = "Welcome Yoll";
        Assert.assertEquals("Welcome message verification failed",
expectedWelcomeMessage, actualWelcomeMessage);
    }

    @Then("framework browser quit")
    public void framework_browser_quit() {
        driver.quit();
    }

    // Additional Scenario:
    @When("user enters empty for username {string} and password {string}")
    public void user_enters_empty_for_username_and_password(String username, String
password) {
        loginPage.usernameInputBox.sendKeys(username);
        loginPage.passwordInputBox.sendKeys(password);
    }

    @Then("user can see error {string}")
    public void user_can_see_error(String expectedErrorMessage) {
        String actualErrorMessage = loginPage.errorMessage.getText();
        Assert.assertEquals("Error message verification failed", expectedErrorMessage,
actualErrorMessage);
    }
}
```

```
@regression
Feature: User Login Process

  Background: Preliminary Steps
    Given user navigates to hrm login page
  @loginValid @smoke
  Scenario: Login with valid credentials
```

```
    #Given user navigates to hrm login page
    When user enters username "yoll-student" and password "Bootcamp5#"
    And user clicks login button
    Then user is redirected to home page
    * framework browser quit

  @loginInvalid
  Scenario: Login with invalid credentials
   # Given user navigates to hrm login page
    When user enters username "invalid" and password "invalid"
    And user clicks login button
    Then user can see error "Invalid Credentials"
    #Then user can see an error message
    * framework browser quit
  @emptyUsername
  Scenario: Login with invalid credentials when username empty
    When user enters empty for username "" and password "invalid"
    And user clicks login button
    Then user can see error "Username cannot be empty"
    #Then user can see an error message
    * framework browser quit

  @emptyPassword
  Scenario: Login with invalid credentials when password empty
    When user enters invalid username "invalid" and password ""
    And user clicks login button
    Then user can see error "Password cannot be empty"
    #Then user can see an error message that says Password cannot be empty
    * framework browser quit

@run @wip # wip: Work in Progress
  Scenario: Test parameters with different data types
    #* test with boolean true
    * test with double 20.92
    * test with int 25
    * test with string "hello"
    * framework browser quit
```

Task First Version:

Write two features files : user-login.feature and user-search.feature

Two java files : LoginSteps.java and SearchSteps.java

TestRunner.java

HomePage.java LoginPage.java

TestDataTypesTest.java for (Double, Integer, String) , the scenario is in user-login.feature

First File: user-login.feature

```
@regression
Feature: User Login Process

  Background: Preliminary Steps
    Given user navigates to hrm login page
  @loginValid @smoke
  Scenario: Login with valid credentials
    #Given user navigates to hrm login page
    When user enters username "yoll-student" and password "Bootcamp5#"
    And user clicks login button
    Then user is redirected to home page
    * framework browser quit

  @loginInvalid
  Scenario: Login with invalid credentials
   # Given user navigates to hrm login page
    When user enters username "invalid" and password "invalid"
    And user clicks login button
    Then user can see error "Invalid Credentials"
    #Then user can see an error message
    * framework browser quit
  @emptyUsername
  Scenario: Login with invalid credentials when username empty
    When user enters empty for username "" and password "invalid"
    And user clicks login button
    Then user can see error "Username cannot be empty"
    #Then user can see an error message
    * framework browser quit

  @emptyPassword
  Scenario: Login with invalid credentials when password empty
    When user enters invalid username "invalid" and password ""
    And user clicks login button
    Then user can see error "Password cannot be empty"
    #Then user can see an error message that says Password cannot be empty
    * framework browser quit

  @run @wip # wip: Work in Progress
  Scenario: Test parameters with different data types
    #* test with boolean true
    * test with double 20.92
    * test with int 25
    * test with string "hello"
    * framework browser quit
```

Second File: user-search.feature

```
Feature: User search functionality

  @run
  Scenario: Search a product
    Given user lands on amazon.com
```

TestRunner.java

```java
package runner;
import io.cucumber.junit.Cucumber;
import io.cucumber.junit. CucumberOptions;
import org.junit.runner. RunWith;


@RunWith(Cucumber.class)
@CucumberOptions(

// first we need to teach our framework where all the features will be located
features="src/test/resources/features",
// then we have to connect the feature steps with step definitions
// for this we will use "glue" to provide the location to step definitions
// to provide location of steps we start wil by default "src/test/java"
glue="step_definitions",

// dryRun will make sure that all the steps have a defined method
// if it is set to tru it won't run the test but just check if all steps are defined
// if set to false it will run the tests regularly
        dryRun = false,
        //tags= "@emptyPassword",
        //tags="@smoke",
        //tags="@run",
        //tags="@loginInvalid and @smoke"
        //tags="@loginInvalid and @regression"
        //tags="@loginInvalid and @loginValid"
        monochrome = true,
        plugin = {"pretty","html:target/reports/cucumber-reports.html" }

)
public class TestRunner {}

// no code need inside of this class!
```

TestDataTypesTest.java

```java
package step_definitions;

import io.cucumber.java.en.Given;

import java.awt.*;

public class TestDataTypesTest {
   /**
    @Given("test with boolean {boolean}")
    public void test_with_boolean_true(Boolean bool) {
        System.out.println(bool);
    }
    */
    @Given("test with double {double}")
    public void test_with_double(Double double1) {
        System.out.println(double1);

    }
    @Given("test with int {int}")
    public void test_with_int(Integer int1) {
        System.out.println(int1);
```

```
    }
    @Given("test with string {string}")
    public void test_with_string(String string) {
        System.out.println(string);

    }
}
```

SearchSteps.java

```
package step_definitions;

import io.cucumber.java.en.Given;
import org.openqa.selenium.WebDriver;
import utilities.DriverFactory;

public class SearchSteps {
    private WebDriver driver = DriverFactory.getDriver ("CHROME");
    @Given("user lands on amazon.com")
    public void user_lands_on_amazon_com() {
        driver.get("https://www.amazon.com");

    }

}
```

LoginSteps.java

```
package step_definitions;

import io.cucumber.java.en.Given;
import io.cucumber.java.en.Then;
import io.cucumber.java.en.When;
import org.junit.Assert;
import org.openqa.selenium.WebDriver;
import utilities.DriverFactory;
import pages.HomePage;
import pages.LoginPage;

public class LoginSteps {

    private WebDriver driver = DriverFactory.getDriver("CHROME");
    private LoginPage loginPage = new LoginPage(driver);
    private HomePage homePage = new HomePage(driver);

    @Given("user navigates to hrm login page")
    public void user_navigates_to_hrm_login_page() {
        driver.get("http://dev-hrm.yoll.io/index.php/auth/login");
    }

    @When("user enters username {string} and password {string}")
    public void user_enters_username_and_password(String username, String password) {
        loginPage.usernameInputBox.sendKeys(username);
        loginPage.passwordInputBox.sendKeys(password);
    }

    @When("user enters invalid username {string} and password {string}")
```

```
    public void user_enters_invalid_username_and_password(String username, String
password) {
        loginPage.usernameInputBox.sendKeys(username);
        loginPage.passwordInputBox.sendKeys(password);
    }

    @When("user enters empty for username {string} and password {string}")
    public void user_enters_empty_for_username_and_password(String username, String
password) {
        loginPage.usernameInputBox.sendKeys(username);
        loginPage.passwordInputBox.sendKeys(password);
    }

    @When("user clicks login button")
    public void user_clicks_login_button() {
        loginPage.loginButton.click();
    }

    @Then("user is redirected to home page")
    public void user_is_redirected_to_home_page() {
        String actualWelcomeMessage = homePage.welcomeMessage.getText();
        String expectedWelcomeMessage = "Welcome Yoll";
        Assert.assertEquals("Welcome message verification failed",
expectedWelcomeMessage, actualWelcomeMessage);
    }

    @Then("user can see error {string}")
    public void user_can_see_error(String expectedErrorMessage) {
        String actualErrorMessage = loginPage.errorMessage.getText();
        Assert.assertEquals("Error message verification failed", expectedErrorMessage,
actualErrorMessage);
    }

    @Then("framework browser quit")
    public void framework_browser_quit() {
        driver.quit();
    }

    // Additional test steps for other data types without duplicates
    @When("test with boolean {string}")
    public void test_with_boolean(String value) {
        boolean booleanValue = Boolean.parseBoolean(value);
        System.out.println("Boolean value: " + booleanValue);
    }

    @When("test with double value {double}")
    public void test_with_double_value(Double value) {
        System.out.println("Double value: " + value);
    }

    @When("test with int value {int}")
    public void test_with_int_value(int value) {
        System.out.println("Int value: " + value);
    }

    @When("test with string value {string}")
    public void test_with_string_value(String value) {
        System.out.println("String value: " + value);
    }
}
```

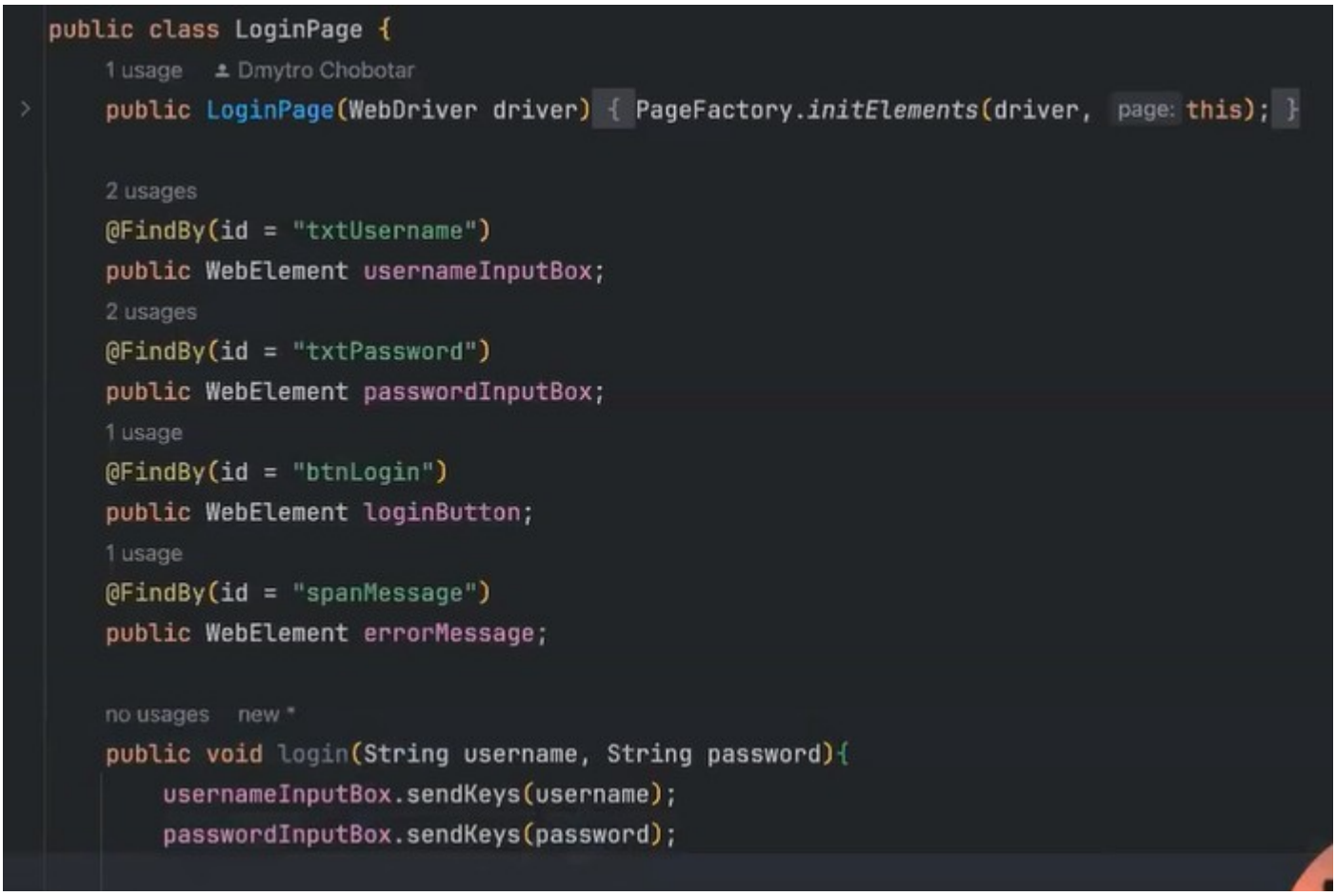
LoginPage.java

```java
package pages;

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.FindBy;
import org.openqa.selenium.support.PageFactory;

/**
 * LoginPage represent real HRM Login Page as a WebElements repository
 * The properties are public so we can access them directly
 * Here PageFactory is used to initialize the WebElements with @FindBy annotations
 **/
public class LoginPage {

    public LoginPage(WebDriver driver) {

        PageFactory.initElements(driver, this);
    }

    @FindBy(id = "txtUsername")
    public WebElement usernameInputBox;
    @FindBy(id = "txtPassword")
    public WebElement passwordInputBox;
    @FindBy(id = "btnLogin")
    public WebElement loginButton;
    @FindBy(id = "spanMessage")
    public WebElement errorMessage;
}
```

HomePage.java

```java
package pages;

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.FindBy;
import org.openqa.selenium.support.PageFactory;
public class HomePage {

    public HomePage(WebDriver driver) {
        PageFactory.initElements(driver,this);

    }
    @FindBy(id = "welcome")
    public WebElement welcomeMessage;
}
```

Task Second Version:



```java
public class LoginPage {
    1 usage  Dmytro Chobotar
    public LoginPage(WebDriver driver) { PageFactory.initElements(driver, page: this); }

    2 usages
    @FindBy(id = "txtUsername")
    public WebElement usernameInputBox;
    2 usages
    @FindBy(id = "txtPassword")
    public WebElement passwordInputBox;
    1 usage
    @FindBy(id = "btnLogin")
    public WebElement loginButton;
    1 usage
    @FindBy(id = "spanMessage")
    public WebElement errorMessage;

    no usages  new *
    public void login(String username, String password){
        usernameInputBox.sendKeys(username);
        passwordInputBox.sendKeys(password);
```

user-login.feature

```
Feature: User Login Process

  Background: Preliminary Steps
    Given user navigates to hrm login page
  @loginValid @smoke
  Scenario: Login with valid credentials
    #Given user navigates to hrm login page
    When user enters username "yoll-student" and password "Bootcamp5#"
    And user clicks login button
    Then user is redirected to home page
    * framework browser quit

  @loginInvalid
  Scenario: Login with invalid credentials
   # Given user navigates to hrm login page
    When user enters username "invalid" and password "invalid"
    And user clicks login button
    Then user can see error "Invalid Credentials"
    #Then user can see an error message
    * framework browser quit
  @emptyUsername
  Scenario: Login with invalid credentials when username empty
    When user enters empty for username "" and password "invalid"
    And user clicks login button
    Then user can see error "Username cannot be empty"
    #Then user can see an error message
```

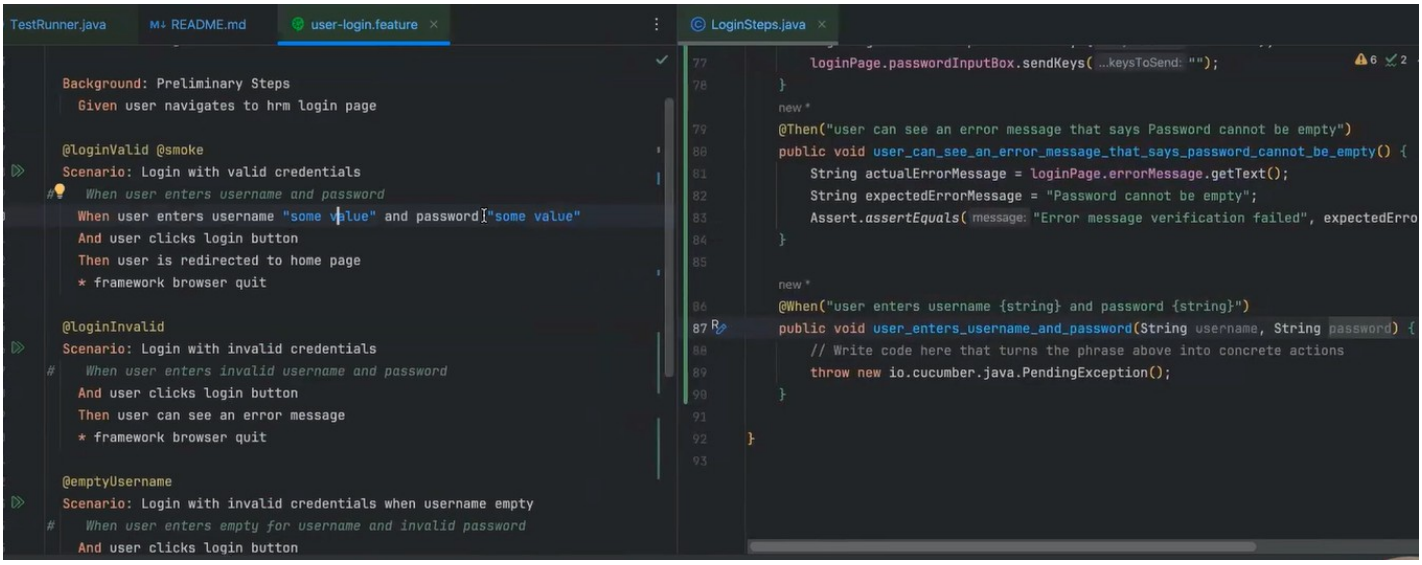
```
    * framework browser quit

  @emptyPassword
  Scenario: Login with invalid credentials when password empty
    When user enters invalid username "invalid" and password ""
    And user clicks login button
    Then user can see error "Password cannot be empty"
    #Then user can see an error message that says Password cannot be empty
    * framework browser quit
```

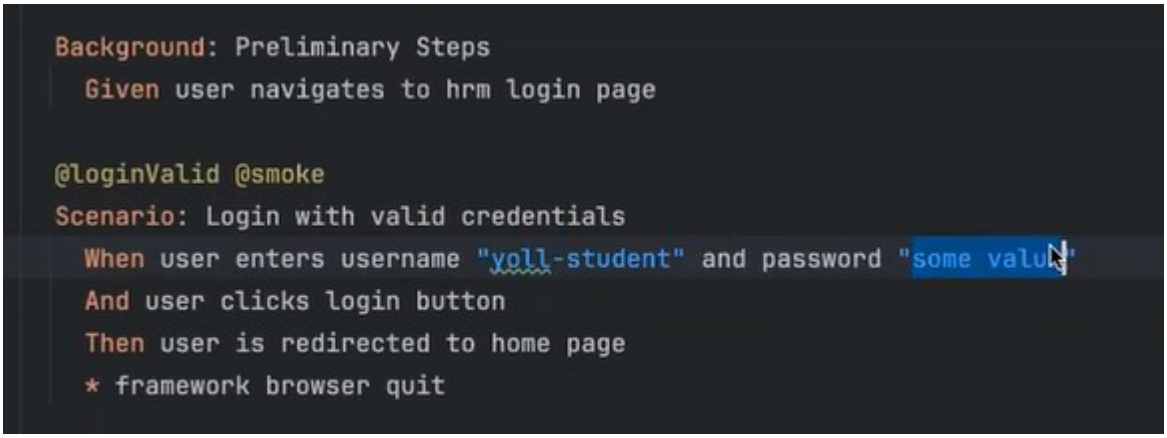
```
package step_definitions;

import io.cucumber.java.en.Given;
import io.cucumber.java.en.Then;
import io.cucumber.java.en.When;
import org.junit.Assert;
import org.openqa.selenium.WebDriver;
import pages.HomePage;
import pages.LoginPage;
import utilities.DriverFactory;

public class LoginSteps {

    private WebDriver driver = DriverFactory.getDriver ("CHROME");

    private LoginPage loginPage = new LoginPage(driver);
    private HomePage homePage = new HomePage(driver);  // Instantiate HomePage
    //First Scenario:

    @Given("user navigates to hrm login page")
    public void user_navigates_to_hrm_login_page() {
        driver.get("http://dev-hrm.yoll.io/index.php/auth/login");
    }
    @When("user enters username and password")
    public void user_enters_username_and_password() {
        loginPage.usernameInputBox.sendKeys("yoll-student");
        loginPage.passwordInputBox.sendKeys( "Bootcamp5#");
    }
    @When("user clicks login button")
    public void user_clicks_login_button() {
        loginPage.loginButton.click();

    }
    @Then("user is redirected to home page")
    public void user_is_redirected_to_home_page() {
        String actualWelcomeMessage = homePage.welcomeMessage.getText();
        String expectedWelcomeMessage = "Welcome Yoll";
// we will need to assert actual and expected
// The testNG structure of assert: Assert.assertEquals (actual, expected, message)
// In JUnit there are identical hard assert methods
        //Assert must be JUnit
        Assert.assertEquals(  "Welcome message verification failed",
expectedWelcomeMessage, actualWelcomeMessage);
```

```
    }
    @Then("framework browser quit")
    public void framework_browser_quit() {
        driver.quit();
    }

    //Second Scenario:
    @When("user enters invalid username and password")
    public void user_enters_invalid_username_and_password() {
        loginPage.usernameInputBox.sendKeys("invalid");
        loginPage.passwordInputBox.sendKeys( "invalid");
    }
    /*
    @Then("user can see an error message")
    public void user_can_see_an_error_message() {
        String actualErrorMessage = loginPage.errorMessage.getText();
        String expectedErrorMessage = "Invalid credentials";
        Assert.assertEquals(  "Error message verification failed",
expectedErrorMessage,actualErrorMessage);
    }
    */
    @When("user enters empty for username and invalid password")
    public void user_enters_empty_for_username_and_invalid_password() {

        loginPage.usernameInputBox.sendKeys("");
        loginPage.passwordInputBox.sendKeys( "invalid");
    }
    /*
    @Then("user can see an error message that says username cannot be empty")
    public void user_can_see_an_error_message_that_says_username_cannot_be_empty() {
        String actualErrorMessage = loginPage.errorMessage.getText();
        String expectedErrorMessage = "Username cannot be empty";
        Assert.assertEquals(  "Error message verification failed",
expectedErrorMessage,actualErrorMessage);
    }
    */
    /*
    @When("user enters invalid username and empty for password")
    public void user_enters_invalid_username_and_empty_for_password() {
        loginPage.usernameInputBox.sendKeys("invalid");
        loginPage.passwordInputBox.sendKeys( "");
    }
    */
    @When("user enters invalid username {string} and password {string}")
    public void user_enters_invalid_username_and_password(String string, String string2)
{
        loginPage.usernameInputBox.sendKeys("invalid");
        loginPage.passwordInputBox.sendKeys( "");
    }
    /*
    @Then("user can see an error message that says Password cannot be empty")
    public void user_can_see_an_error_message_that_says_password_cannot_be_empty() {
        String actualErrorMessage = loginPage.errorMessage.getText();
        String expectedErrorMessage = "Password cannot be empty";
        Assert.assertEquals(  "Error message verification failed",
expectedErrorMessage,actualErrorMessage);
    }
    */
```

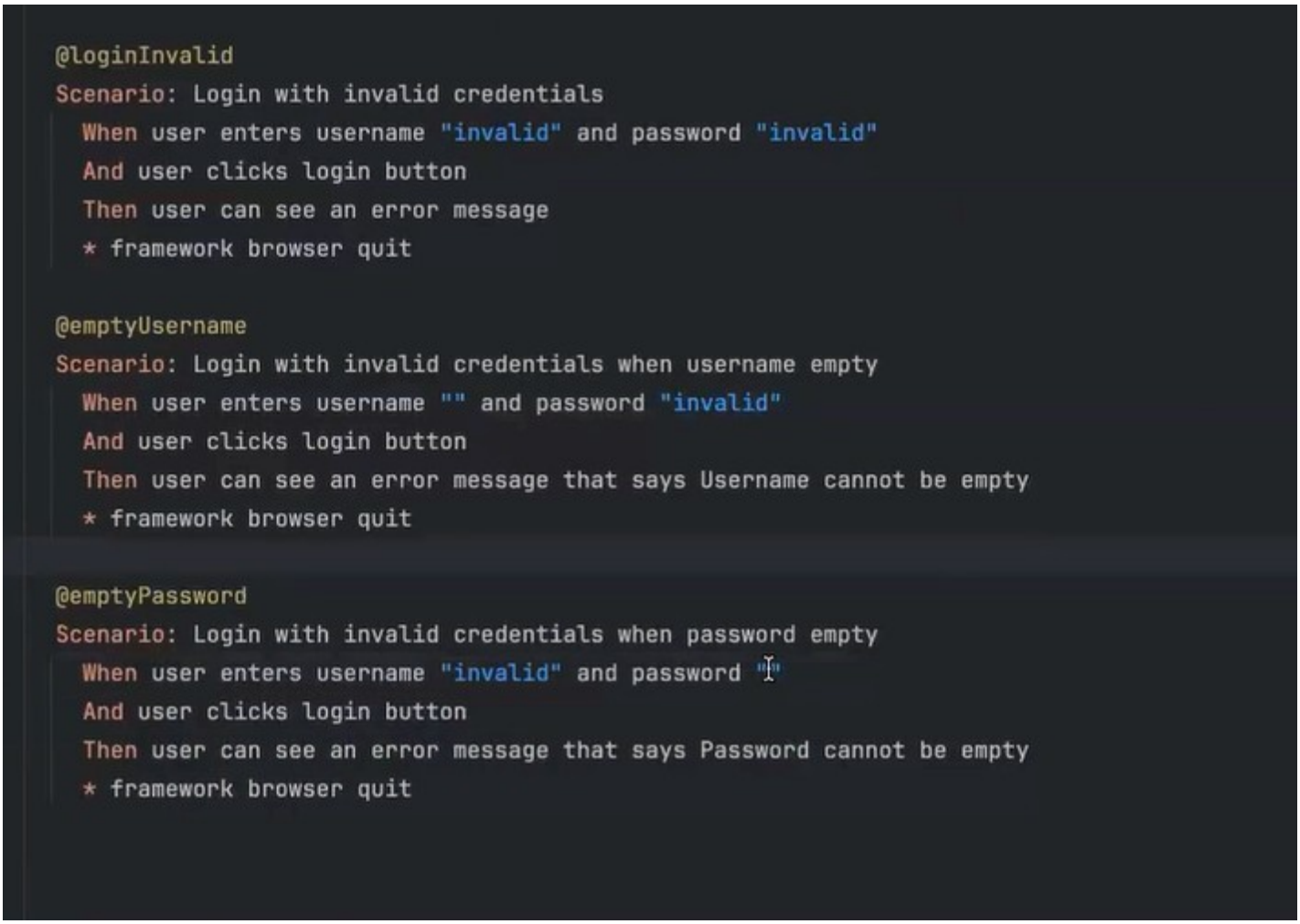
```
    @When("user enters empty for username {string} and password {string}")
    public void user_enters_empty_for_username_and_password(String username, String
password) {
        loginPage.usernameInputBox.sendKeys(username);
        loginPage.passwordInputBox.sendKeys(password);
    }

    @Then("user can see error {string}")
    public void user_can_see_error(String expectedErrorMessage) {
        String actualErrorMessage = loginPage.errorMessage.getText();
        Assert.assertEquals(  "Error message verification failed",
expectedErrorMessage,actualErrorMessage);

    }

    @When("user enters username {string} and password {string}")
    public void userEntersUsernameAndPassword(String arg0, String arg1) {
    }
}
```

Task Third Version


TestRunner.java
README.md
user-login.feature
LoginSteps.java
Background: Preliminary Steps
Given user navigates to hrm login page
@LoginValid @smoke
Scenario: Login with valid credentials
When user enters username and password
When user enters username "some value" and password "some value"
And user clicks login button
Then user is redirected to home page
* framework browser quit
@LoginInvalid
Scenario: Login with invalid credentials
When user enters invalid username and password
And user clicks login button
Then user can see an error message
* framework browser quit
@emptyUsername
Scenario: Login with invalid credentials when username empty
When user enters empty for username and invalid password
And user clicks login button
loginPage.passwordInputBox.sendKeys(...keysToSend: "");
@Then("user can see an error message that says Password cannot be empty")
public void user_can_see_an_error_message_that_says_password_cannot_be_empty() {
String actualErrorMessage = loginPage.errorMessage.getText();
String expectedErrorMessage = "Password cannot be empty";
Assert.assertEquals(message: "Error message verification failed", expectedErro
@When("user enters username {string} and password {string}")
public void user_enters_username_and_password(String username, String password) {
// Write code here that turns the phrase above into concrete actions
throw new io.cucumber.java.PendingException();



```
Background: Preliminary Steps
  Given user navigates to hrm login page

@loginValid @smoke
Scenario: Login with valid credentials
  When user enters username "yoll-student" and password "some valu"
  And user clicks login button
  Then user is redirected to home page
  * framework browser quit
```



```
@loginInvalid
Scenario: Login with invalid credentials
  When user enters username "invalid" and password "invalid"
  And user clicks login button
  Then user can see an error message
  * framework browser quit

@emptyUsername
Scenario: Login with invalid credentials when username empty
  When user enters username "" and password "invalid"
  And user clicks login button
  Then user can see an error message that says Username cannot be empty
  * framework browser quit

@emptyPassword
Scenario: Login with invalid credentials when password empty
  When user enters username "invalid" and password ""
  And user clicks login button
  Then user can see an error message that says Password cannot be empty
  * framework browser quit
```

user-login.feature

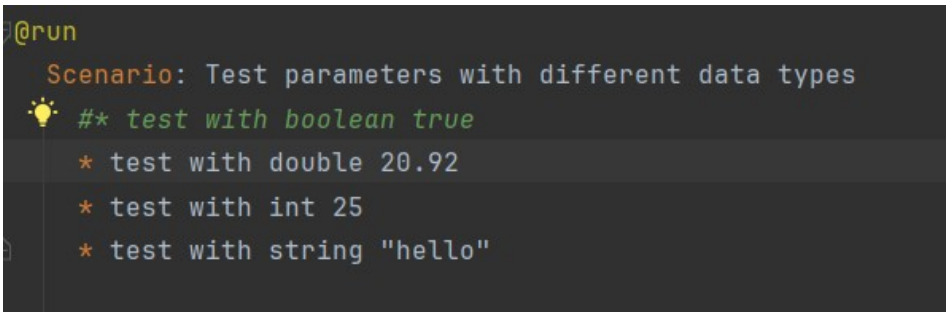
```
@regression
Feature: User Login Process

  Background: Preliminary Steps
    Given user navigates to hrm login page
  @loginValid @smoke
  Scenario: Login with valid credentials
    When user enters username "yoll-student" and password "Bootcamp5#"
    Then user is redirected to home page
    * framework browser quit
```

```
  @loginInvalid
  Scenario: Login with invalid credentials
    When user enters username "invalid" and password "invalid"
    Then user can see error "Invalid Credentials"
    * framework browser quit
  @emptyUsername
  Scenario: Login with invalid credentials when username empty
    When user enters empty for username "" and password "invalid"
    Then user can see error "Username cannot be empty"
    * framework browser quit

  @emptyPassword
  Scenario: Login with invalid credentials when password empty
    When user enters invalid username "invalid" and password ""
    Then user can see error "Password cannot be empty"
.    * framework browser quit

  @run @wip # wip: Work in Progress
  Scenario: Test parameters with different data types
    #* test with boolean true
    * test with double 20.92
    * test with int 25
    * test with string "hello"
    * framework browser quit



package step_definitions;

import io.cucumber.java.en.Given;
import io.cucumber.java.en.Then;
import io.cucumber.java.en.When;
import org.junit.Assert;
import org.openqa.selenium.WebDriver;
import pages.HomePage;
import pages.LoginPage;
import utilities.DriverFactory;

public class LoginSteps {

    private WebDriver driver = DriverFactory.getDriver ("CHROME");

    private LoginPage loginPage = new LoginPage(driver);
    private HomePage homePage = new HomePage(driver);  // Instantiate HomePage
    //First Scenario:

    @Given("user navigates to hrm login page")
    public void user_navigates_to_hrm_login_page() {
        driver.get("http://dev-hrm.yoll.io/index.php/auth/login");
    }
    @When("user enters username and password")
    public void user_enters_username_and_password() {
        loginPage.usernameInputBox.sendKeys("yoll-student");
        loginPage.passwordInputBox.sendKeys( "Bootcamp5#");
    }
    @When("user clicks login button")
    public void user_clicks_login_button() {
        loginPage.loginButton.click();

    }
    @Then("user is redirected to home page")
```

```
    public void user_is_redirected_to_home_page() {
        String actualWelcomeMessage = homePage.welcomeMessage.getText();
        String expectedWelcomeMessage = "Welcome Yoll";
// we will need to assert actual and expected
// The testNG structure of assert: Assert.assertEquals (actual, expected, message)
// In JUnit there are identical hard assert methods
        //Assert must be JUnit
        Assert.assertEquals(  "Welcome message verification failed",
expectedWelcomeMessage, actualWelcomeMessage);

    }
    @Then("framework browser quit")
    public void framework_browser_quit() {
        driver.quit();
    }


    @When("user logs in with or username {string} and password {string}")
    public void user_enters_username_and_password(String username, String password) {
        loginPage.usernameInputBox.sendKeys(username);
        loginPage.passwordInputBox.sendKeys(password);
    }
    @Then("user can see error {string}")
    public void user_can_see_error(String expectedErrorMessage) {
        String actualErrorMessage = loginPage.errorMessage.getText();
        Assert.assertEquals(  "Error message verification failed",
expectedErrorMessage,actualErrorMessage);

    }
}


package pages;

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.FindBy;
import org.openqa.selenium.support.PageFactory;

/**
 * LoginPage represent real HRM Login Page as a WebElements repository
 * The properties are public so we can access them directly
 * Here PageFactory is used to initialize the WebElements with @FindBy annotations
 **/
public class LoginPage {

    public LoginPage(WebDriver driver) {

        PageFactory.initElements(driver, this);
    }

    @FindBy(id = "txtUsername")
    public WebElement usernameInputBox;
    @FindBy(id = "txtPassword")
    public WebElement passwordInputBox;
    @FindBy(id = "btnLogin")
    public WebElement loginButton;
    @FindBy(id = "spanMessage")
    public WebElement errorMessage;

public void login(String username, String password) {
```

```
        usernameInputBox.sendKeys(username);
        passwordInputBox.sendKeys(password);

}
}
```

Task : Test with double, Integer, and String.


@run
Scenario: Test parameters with different data types
#* test with boolean true
* test with double 20.92
* test with int 25
* test with string "hello"

TestRunner java class:

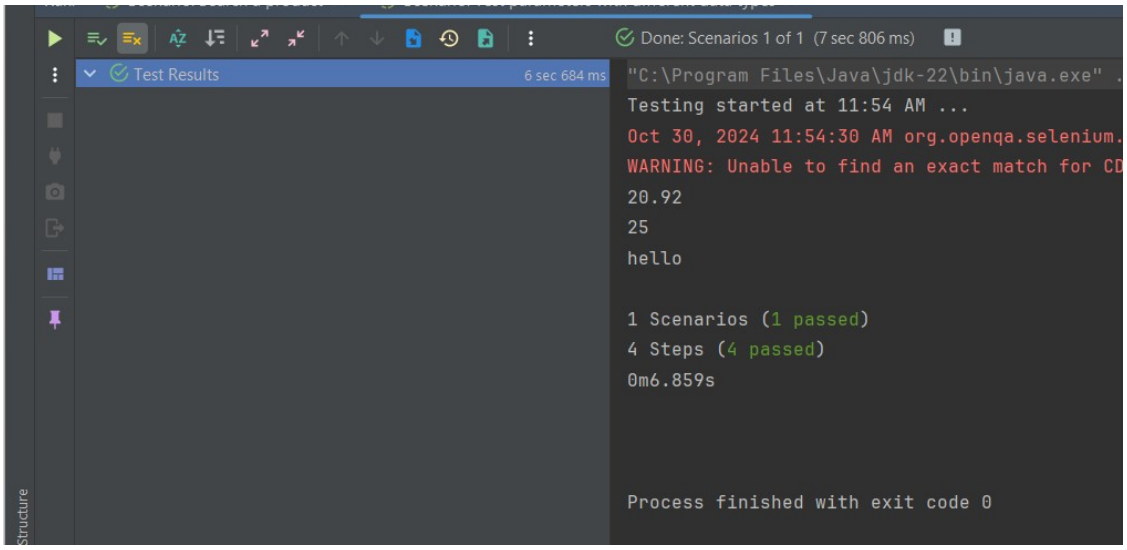
```
package runner;
import io.cucumber.junit.Cucumber;
import io.cucumber.junit. CucumberOptions;
import org.junit.runner. RunWith;


@RunWith(Cucumber.class)
@CucumberOptions(

// first we need to teach our framework where all the features will be located
features="src/test/resources/features",
// then we have to connect the feature steps with step definitions
// for this we will use "glue" to provide the location to step definitions
// to provide location of steps we start wil by default "src/test/java"
glue="step_definitions",

// dryRun will make sure that all the steps have a defined method
// if it is set to tru it won't run the test but just check if all steps are defined
// if set to false it will run the tests regularly
        dryRun = false,
        tags="@run",
        monochrome = true,
        plugin = {"pretty","html:target/reports/cucumber-reports.html" }
```

```
)
public class TestRunner {}

// no code need inside of this class!
```

TestDataTypesTest java class

```
package step_definitions;

import io.cucumber.java.en.Given;

import java.awt.*;

public class TestDataTypesTest {

    @Given("test with double {double}")
    public void test_with_double(Double double1) {
        System.out.println(double1);

    }
    @Given("test with int {int}")
    public void test_with_int(Integer int1) {
        System.out.println(int1);

    }
    @Given("test with string {string}")
    public void test_with_string(String string) {
        System.out.println(string);

    }
}
```

Result:



HTML REPORT

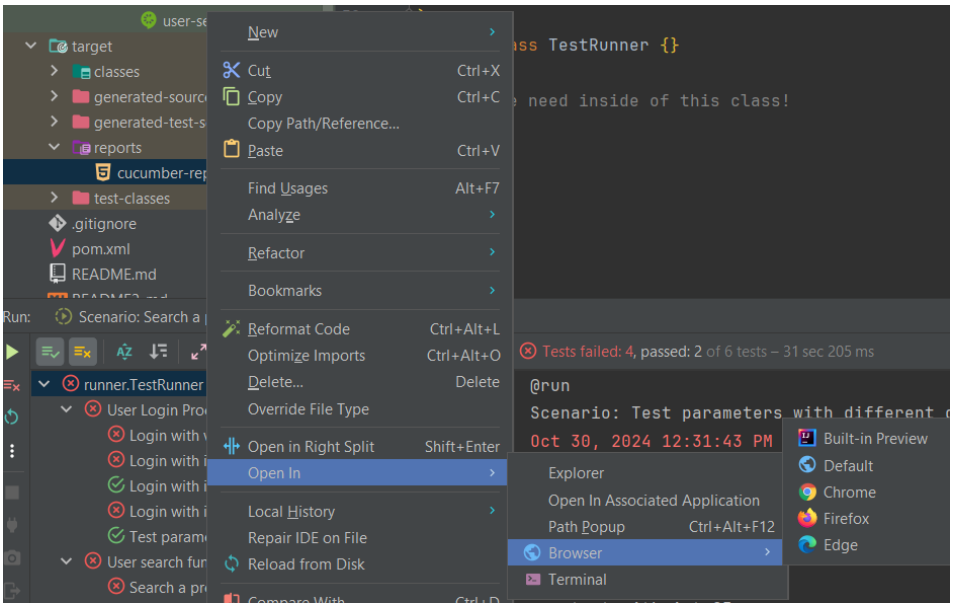
Cucumber provides reports in following formats:

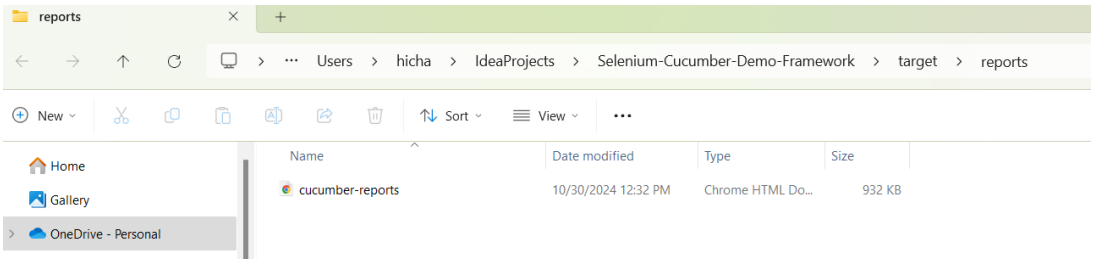
- HTML, XML, JSON

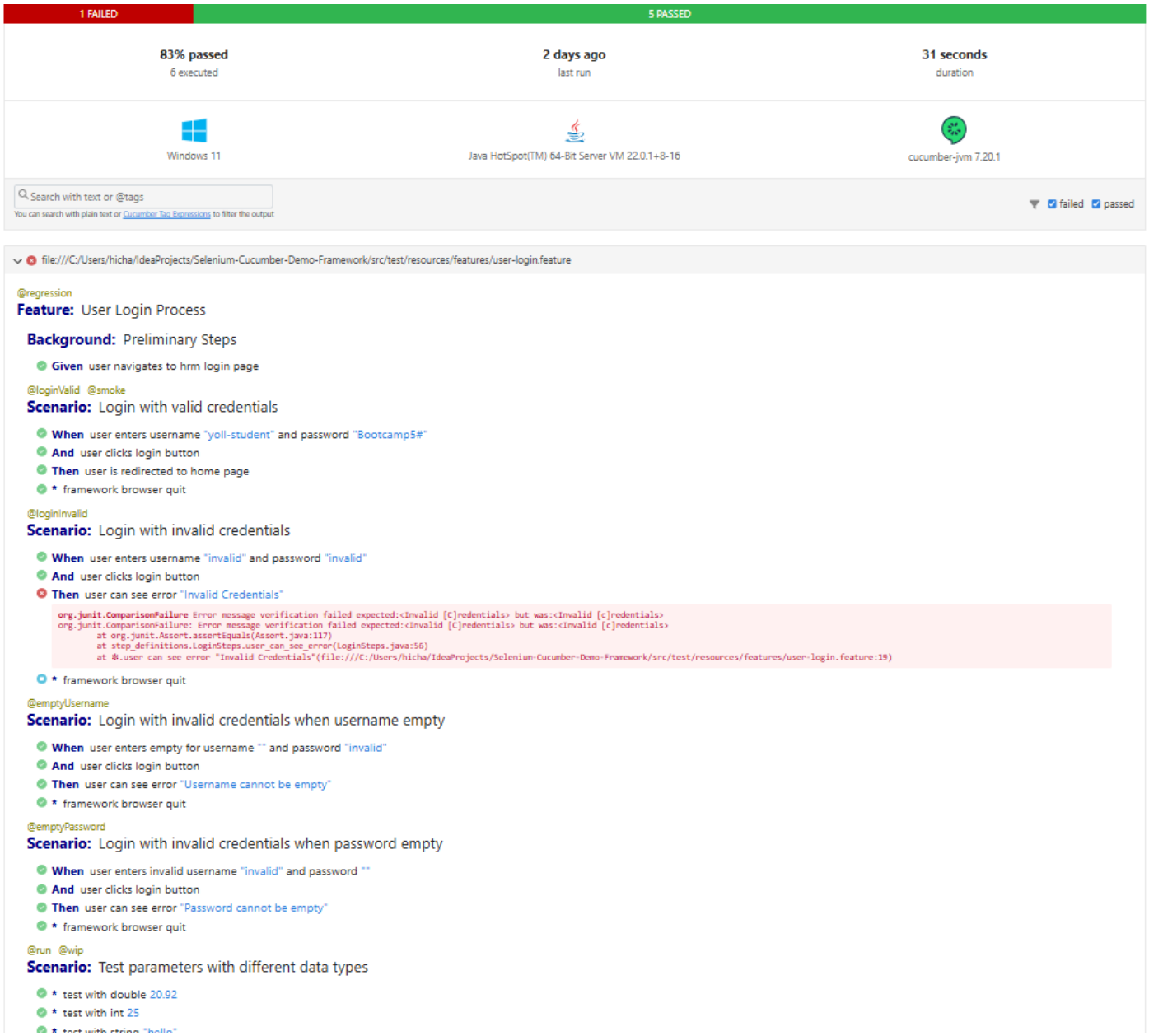
To get HTML report we can use pretty report option in Runner file

html:target/reports/cucumber-reports.html

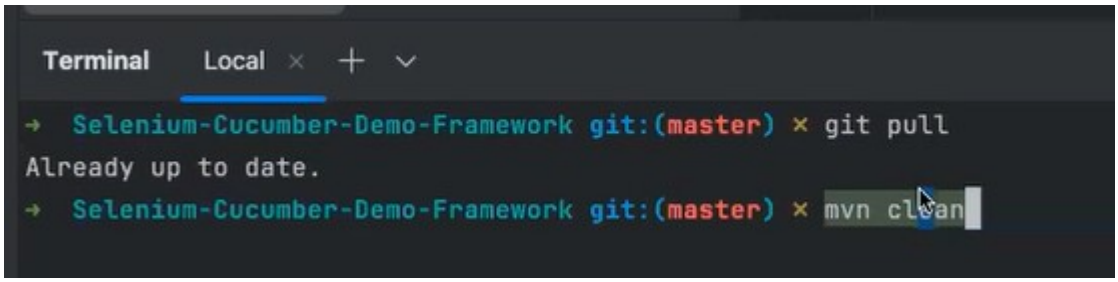
Go to target folder, then open reports folder, open the html file by using the browsers on the left og IntelliJ, or open it in borwser or explorer:


target
classes
reports
cucumber-rep
test-classes
.gitignore
pom.xml
README.md
New
Cut Ctrl+X
Copy Ctrl+C
Copy Path/Reference...
Paste Ctrl+V
Find Usages Alt+F7
Analyze
Refactor
Bookmarks
Reformat Code Ctrl+Alt+L
Optimize Imports Ctrl+Alt+O
Delete... Delete
Override File Type
Open in Right Split Shift+Enter
Open In
Local History
Repair IDE on File
Reload from Disk
Explorer
Open In Associated Application
Path Popup Ctrl+Alt+F12
Browser
Terminal
Built-in Preview
Default
Chrome
Firefox
Edge
TestRunner {}
need inside of this class!
Tests failed: 4, passed: 2 of 6 tests – 31 sec 205 ms
@run
Oct 30, 2024 12:31:43 PM


reports
Users
hicha
IdeaProjects
Selenium-Cucumber-Demo-Framework
target
reports
New
Sort
View
Home
Gallery
OneDrive - Personal
Name
Date modified
Type
Size
cucumber-reports
10/30/2024 12:32 PM
Chrome HTML Do...
932 KB


1 FAILED
5 PASSED
83% passed
6 executed
2 days ago
last run
31 seconds
duration
Windows 11
Java HotSpot(TM) 64-Bit Server VM 22.0.1+8-16
cucumber-jvm 7.20.1
Search with text or @tags
You can search with plain text or Cucumber Tag Expressions to filter the output
failed
passed
file:///C:/Users/hicha/IdeaProjects/Selenium-Cucumber-Demo-Framework/src/test/resources/features/user-login.feature
@regression
Feature: User Login Process
Background: Preliminary Steps
Given user navigates to hrm login page
@loginValid @smoke
Scenario: Login with valid credentials
When user enters username "yoll-student" and password "Bootcamp5#"
And user clicks login button
Then user is redirected to home page
* framework browser quit
@loginInvalid
Scenario: Login with invalid credentials
When user enters username "invalid" and password "invalid"
And user clicks login button
Then user can see error "Invalid Credentials"
org.junit.ComparisonFailure Error message verification failed expected:<Invalid [C]redentials> but was:<Invalid [c]redentials>
org.junit.ComparisonFailure: Error message verification failed expected:<Invalid [C]redentials> but was:<Invalid [c]redentials>
at org.junit.Assert.assertEquals(Assert.java:117)
at step_definitions.LoginSteps.user_can_see_error(LoginSteps.java:56)
at ✽.user can see error "Invalid Credentials"(file:///C:/Users/hicha/IdeaProjects/Selenium-Cucumber-Demo-Framework/src/test/resources/features/user-login.feature:19)
* framework browser quit
@emptyUsername
Scenario: Login with invalid credentials when username empty
When user enters empty for username "" and password "invalid"
And user clicks login button
Then user can see error "Username cannot be empty"
* framework browser quit
@emptyPassword
Scenario: Login with invalid credentials when password empty
When user enters invalid username "invalid" and password ""
And user clicks login button
Then user can see error "Password cannot be empty"
* framework browser quit
@run @wip
Scenario: Test parameters with different data types
* test with double 20.92
* test with int 25

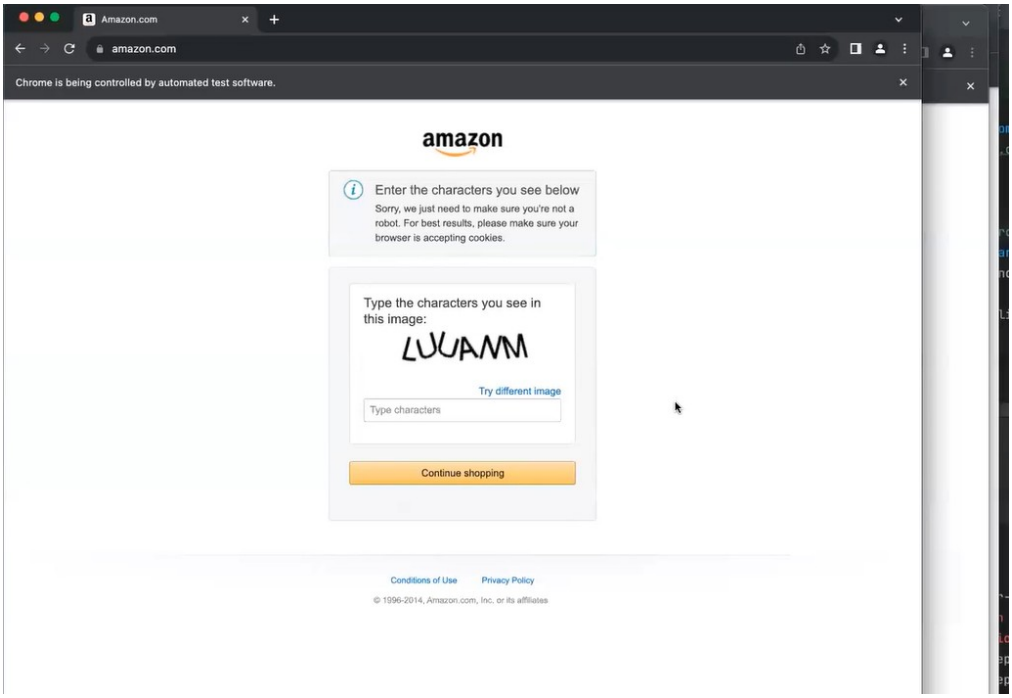
Cleaning : Terminal/Local : mvn clean



Captcha

YOU DO NOT NEED TO DEAL WITH CAPTCHA

IT IS NOT AUTOMATED


Amazon.com
amazon.com
Chrome is being controlled by automated test software.
amazon
Enter the characters you see below
Sorry, we just need to make sure you're not a robot. For best results, please make sure your browser is accepting cookies.
Type the characters you see in this image:
LUUAMM
Try different image
Type characters
Continue shopping
Conditions of Use
Privacy Policy
© 1996-2014, Amazon.com, Inc. or its affiliates

Task : BestBuy

Feature File

```
Feature: User search functionality

  @bestbuyRun
  Scenario Outline: Search a product
    Given user lands on best buy
    When user enters "<text>" in search box
    * framework browser quit
    @smartphones
    Examples:
      |text                        |
      |iPhone 13 Pro MAX           |
      |Samsung Galaxy Case         |
      |iPhone 15 Pro Max           |
      |iPhone 11 Pro MAX         |
    @laptops
    Examples:
      |text                |
      |MackBook Pro        |
      |MackBook M2         |
      |MackBook Air 2023   |
```

|Gaming Laptop |

BestBuySearchSteps java File

```
package step_definitions;

import io.cucumber.java.en.Given;
import io.cucumber.java.en.When;
import org.openqa.selenium.WebDriver;
import pages.BestBuyLandingPage;
import utilities.DriverFactory;

public class BestBuySearchSteps {
    private WebDriver driver = DriverFactory.getDriver ("CHROME");
    private BestBuyLandingPage BestBuyLandingPage = new
BestBuyLandingPage(driver);
    @Given("user lands on best buy")
    public void user_lands_on_best_buy() {
        driver.get("https://www.bestbuy.com");
    }
    @When("user enters {string} in search box")
    public void user_enters_in_search_box(String text) throws InterruptedException
{
        BestBuyLandingPage.searchBox.sendKeys(text);
        BestBuyLandingPage.searchIcon.click();
        //Thread.sleep(5000);
        //BestBuyLandingPage.searchIcon.sendKeys(text);
        //Thread.sleep(5000);
    }
}
```

BestBuyLandingSteps java File

```
package pages;

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.FindBy;
import org.openqa.selenium.support.PageFactory;

public class BestBuyLandingPage {
    public BestBuyLandingPage(WebDriver driver) {
        PageFactory.initElements(driver,this);
}
@FindBy(id="gh-search-input")
    public WebElement searchBox;
    @FindBy(css="[title='submit search']")
    public WebElement searchIcon;
}
```

TestRunner java File

```
package runner;
import io.cucumber.junit.Cucumber;
import io.cucumber.junit. CucumberOptions;
import org.junit.runner. RunWith;


@RunWith(Cucumber.class)
@CucumberOptions(

// first we need to teach our framework where all the features will be located
features="src/test/resources/features",
// then we have to connect the feature steps with step definitions
// for this we will use "glue" to provide the location to step definitions
// to provide location of steps we start wil by default "src/test/java"
glue="step_definitions",

// dryRun will make sure that all the steps have a defined method
// if it is set to tru it won't run the test but just check if all steps are
defined
// if set to false it will run the tests regularly
        dryRun = false,
        tags= "@laptops",
        //tags="@smoke",
        //tags="@run",
        //tags="@loginInvalid and @smoke"
        //tags="@loginInvalid and @regression"
        //tags="@loginInvalid and @loginValid"
        monochrome = true,
        plugin = {"pretty","html:target/reports/cucumber-reports.html" }

)


public class TestRunner {}

// no code need inside of this class!
```

List, Map, List Of Maps, List of Lists:

In Cucumber, `List` and `Map` are commonly used in step definitions to capture and manipulate tabular data passed from Gherkin scenarios.

List

1. Using `List` in Cucumber

You can use `List` when working with a single column of values. In Gherkin, you would define a table with one column, and Cucumber will map it to a Java `List`.

Example Scenario with `List`

gherkin

```
Feature: Fruit selection

  Scenario: Select fruits
    Given I have the following fruits:
      | Fruit     |
      | Apple     |
      | Banana    |
      | Orange    |
```

Step Definition with `List<String>`

```
import io.cucumber.java.en.Given;
import java.util.List;

public class FruitStepDefinitions {

    @Given("I have the following fruits:")
    public void i_have_the_following_fruits(List<String> fruits) {
        System.out.println("Fruits: " + fruits);
        // Here you can use the list of fruits as needed.
    }
}
```

In this case, `fruits` is a `List<String>` that will contain `[Apple, Banana, Orange]`.

Map

2. Using `Map` in Cucumber

A `Map` is useful when your table has two columns, typically representing key-value pairs.

Example Scenario with `Map`

```
gherkin

Feature: User registration

  Scenario: Register a user
    Given I enter the following details:
      | Field        | Value          |
      | Username     | johndoe        |
      | Password     | password123    |
      | Email        | john@example.com |
```

Step Definition with `Map<String, String>`

```
import io.cucumber.java.en.Given;
import java.util.Map;

public class UserRegistrationStepDefinitions {

    @Given("I enter the following details:")
    public void i_enter_the_following_details(Map<String, String> userDetails) {
        System.out.println("User Details: " + userDetails);
```

```
        // You can access individual fields, for example:
        String username = userDetails.get("Username");
        String password = userDetails.get("Password");
        String email = userDetails.get("Email");

        // Use these values as needed
    }
}
```

In this case, `userDetails` is a `Map<String, String>` with key-value pairs such as `{"Username"="johndoe", "Password"="password123", "Email"="john@example.com"}`.

List of Lists

In Cucumber, a **List of Lists** can be used to handle tabular data when you need to work with multiple rows and columns in Gherkin tables. Cucumber can automatically convert Gherkin tables to a `List<List<String>>` format in your step definitions.

Example: Using `List<List<String>>` in Cucumber

Suppose you want to pass a table of fruits with their categories and prices. Here's how you can set it up in Gherkin and handle it in your step definition.

Feature File (Gherkin Syntax)

```
gherkin
Feature: Fruit inventory

  Scenario: Display list of fruits
    Given I have the following fruit inventory:
      | Fruit   | Category | Price |
      | Apple   | Fruit    | 1.50  |
      | Banana  | Fruit    | 0.75  |
      | Carrot  | Vegetable| 0.30  |
```

In this table:

- Each row is a list, where each cell in the row is a `String`.
- Cucumber will convert the whole table into a `List<List<String>>`.

Step Definition with `List<List<String>>`

```
import io.cucumber.java.en.Given;
import java.util.List;

public class FruitInventoryStepDefinitions {

    @Given("I have the following fruit inventory:")
    public void i_have_the_following_fruit_inventory(List<List<String>> table) {
        System.out.println("--- Fruit Inventory ---");
```

```
        // Print each row in the table
        for (List<String> row : table) {
            System.out.println(row);
        }

        // Access specific elements if needed
        String fruit = table.get(1).get(0);   // Apple
        String category = table.get(1).get(1); // Fruit
        String price = table.get(1).get(2);    // 1.50
        System.out.println("First Fruit Entry: " + fruit + " - " + category + " - $" +
price);
    }
}
```

Explanation

1. **Using `List<List<String>>`**: Cucumber automatically passes the table to the step definition as a `List<List<String>>`. Each row in the Gherkin table becomes a list of strings.
2. **Iterating Through Rows and Columns**:
    - The outer `List` represents rows, and each inner `List` represents the columns in that row.
    - In the example above, each `row` contains `[Fruit, Category, Price]` for each item.
3. **Accessing Specific Cells**:
    - You can use nested `.get()` calls to access specific cells, such as `table.get(rowIndex).get(columnIndex)`.
    - Here, `table.get(1).get(0)` accesses the first column of the second row, which would return `"Apple"`.

Output Example

When running the scenario, the output will look something like this:

```
--- Fruit Inventory ---
[Fruit, Category, Price]
[Apple, Fruit, 1.50]
[Banana, Fruit, 0.75]
[Carrot, Vegetable, 0.30]
First Fruit Entry: Apple - Fruit - $1.50
```

Notes

- The first row in the Gherkin table (headers) will also be included in the `List<List<String>>`. To skip it, you can start iterating from `table.get(1)`.
- This approach works well when you don't need to reference columns by name. If column names are required, consider using a `List<Map<String, String>>`, which is often more flexible.

Summary

- Use `List<List<String>>` to handle tables in Gherkin when working with simple tabular data.

- Access rows and cells using nested `.get()` calls.
- Iterate through rows and columns for displaying or processing data.

List of Maps:

In Cucumber, you can use a **List of Maps** to handle tables with multiple rows, where each row represents a map of key-value pairs. This structure (`List<Map<String, String>>`) is useful when each row has specific column headers, which can act as keys in the map.

Example: Using a `List<Map<String, String>>` in Cucumber

Suppose you want to pass a table of users with fields like `Username`, `Role`, and `Email`. You can represent each user as a map and have a list of these maps to handle multiple users.

Feature File (Gherkin Syntax)

```
gherkin
Copy code
Feature: User management

  Scenario: Add multiple users
    Given I have the following users:
      | Username | Role  | Email               |
      | alice    | admin | alice@example.com   |
      | bob      | user  | bob@example.com     |
      | charlie  | guest | charlie@example.com |
```

In this table:

- Each row corresponds to a `Map<String, String>`, where column headers (`Username`, `Role`, `Email`) are keys, and cell values are the map's values.
- The list of these maps represents the entire table.

Step Definition with `List<Map<String, String>>`

```
java
Copy code
import io.cucumber.java.en.Given;
import java.util.List;
import java.util.Map;

public class UserManagementStepDefinitions {

    @Given("I have the following users:")
    public void i_have_the_following_users(List<Map<String, String>> users) {
        System.out.println("--- User Details ---");

        // Iterate through each user (each map in the list)
        for (Map<String, String> user : users) {
            System.out.println("Username: " + user.get("Username"));
```

```
            System.out.println("Role: " + user.get("Role"));
            System.out.println("Email: " + user.get("Email"));
            System.out.println("-----------");
        }

        // Access individual elements if needed
        String firstUser = users.get(0).get("Username");   // alice
        String firstUserRole = users.get(0).get("Role");   // admin
        System.out.println("First user: " + firstUser + " - Role: " + firstUserRole);
    }
}
```

Explanation

1. **Using `List<Map<String, String>>`**:
    - Cucumber converts each row of the Gherkin table into a `Map<String, String>`, with column headers as keys and cell values as the map values.
    - The whole table is then represented as a `List` of these maps.
2. **Iterating Through the List of Maps**:
    - The outer `List` represents all the rows in the table, while each `Map<String, String>` within the list represents a row.
    - You can access each user's details (or row data) through the map's `get()` method, using the column header as the key.
3. **Accessing Specific Rows and Values**:
    - Access a specific row in the list using `users.get(rowIndex)` and a specific column in that row with `user.get("ColumnName")`.

Output Example

When running the scenario, the output might look like this:

```
markdown
Copy code
--- User Details ---
Username: alice
Role: admin
Email: alice@example.com
-----------
Username: bob
Role: user
Email: bob@example.com
-----------
Username: charlie
Role: guest
Email: charlie@example.com
-----------
First user: alice - Role: admin
```

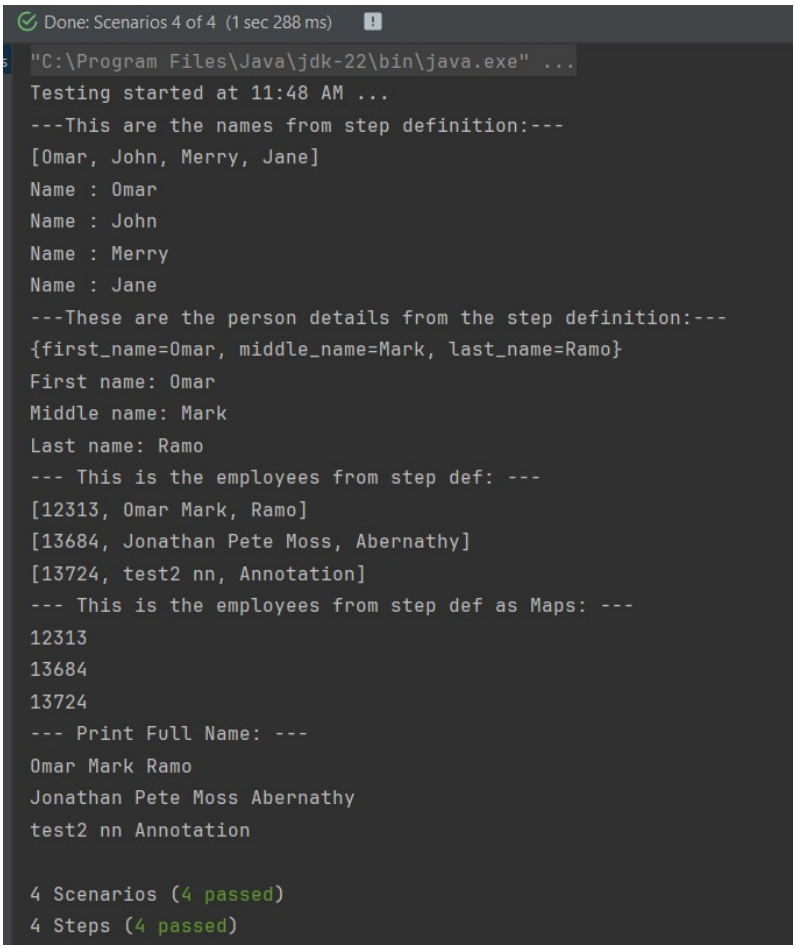
Summary

- Use `List<Map<String, String>>` when you need to work with a table in Gherkin where each row represents a set of key-value pairs.
- Each map in the list represents a row, where the keys are the column headers.
- You can iterate through the list to access each map or access specific values by using `get()` calls with the column header names.

This structure is highly flexible and ideal for tables with column headers where each row represents a different entity, such as a list of users, products, or any other items with structured attributes.

```
@listMap
Feature: Demo the different ways of data tables in cucumber
  The Data tables will be used when we need to provide more than one data for a step
  In cases when we need to verify multiple points on a page at once with one launch to
web page of the user
  The data can be passed as:
  - List<String>
  Map<String, String>
  List<List<String>>
  List<Map<String, String>>
  Custom Type
# In cucumber to describe a test case we use Scenario keyword or we could use as well
Example keyword
# They are synonyms in cucumber - they are pretty much the same

  @listDataTable
  Example: Data table as a list
    Given this is a data table as a list
      | Omar |
      | John |
      | Merry |
      | Jane |
   @mapDataTable
   Example: Data table as a list
     Given this is a data table as a map
       | first_name | Omar|
       | middle_name| Mark|
       | last_name  | Ramo|
  @listOfListsDataTable
  Example: Data Table as a List of Lists
    Given this is a data table as a list of Lists
  | 12313 | Omar Mark |  Ramo|
  | 13684 | Jonathan Pete Moss | Abernathy |
  | 13724 | test2 nn           | Annotation|

  @listOfMapsDataTable
  Example: Data Table as a List of Maps
    Given this is a data table as a list of Maps
      | id    | first_name         |last_name  |
      | 12313 | Omar Mark          |  Ramo     |
      | 13684 | Jonathan Pete Moss | Abernathy |
      | 13724 | test2 nn           | Annotation|
```

After running : @listMap from feature file, the result is:


Done: Scenarios 4 of 4 (1 sec 288 ms)
"C:\Program Files\Java\jdk-22\bin\java.exe" ...
Testing started at 11:48 AM ...
---This are the names from step definition:---
[Omar, John, Merry, Jane]
Name : Omar
Name : John
Name : Merry
Name : Jane
---These are the person details from the step definition:---
{first_name=Omar, middle_name=Mark, last_name=Ramo}
First name: Omar
Middle name: Mark
Last name: Ramo
--- This is the employees from step def: ---
[12313, Omar Mark, Ramo]
[13684, Jonathan Pete Moss, Abernathy]
[13724, test2 nn, Annotation]
--- This is the employees from step def as Maps: ---
12313
13684
13724
--- Print Full Name: ---
Omar Mark Ramo
Jonathan Pete Moss Abernathy
test2 nn Annotation
4 Scenarios (4 passed)
4 Steps (4 passed)

DataTableExampleSteps java file

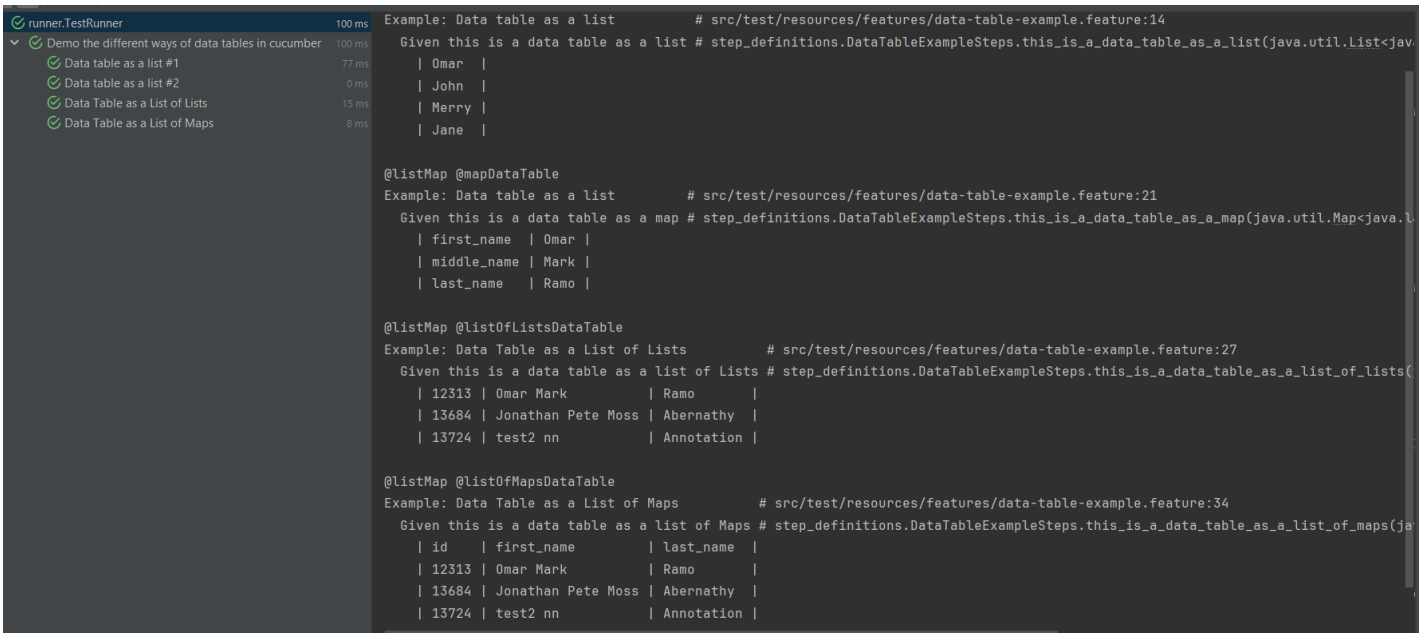
```
package step_definitions;

import io.cucumber.java.en.Given;

import java.util.List;
import java.util.Map;

public class DataTableExampleSteps {
    @Given("this is a data table as a list")//List
    public void this_is_a_data_table_as_a_list(List<String> names) {
        System.out.println("---This are the names from step definition:---" );
        System.out.println(names);

        for (String name : names) {
            System.out.println("Name : " + name);
        }
    }
    @Given("this is a data table as a map")
    public void this_is_a_data_table_as_a_map(Map<String, String> person) {
        System.out.println("---These are the person details from the step
definition:---");
        System.out.println(person);

        // Now you can get any value from the map
        System.out.println("First name: " + person.get("first_name"));
        System.out.println("Middle name: " + person.get("middle_name"));
```

```
            System.out.println("Last name: " + person.get("last_name"));
        }

        @Given("this is a data table as a list of Lists")
        public void this_is_a_data_table_as_a_list_of_lists(List<List<String>>
employees) {
            System.out.println("--- This is the employees from step def: ---");
            for (List<String> employee: employees) {
                System.out.println(employee);
        }
        }

        @Given("this is a data table as a list of Maps")
        public void this_is_a_data_table_as_a_list_of_maps(List<Map<String, String>>
employees) {
            System.out.println("--- This is the employees from step def as Maps:
---");
            for (Map<String, String> map: employees) {
                System.out.println(map.get("id"));
            }
            System.out.println("--- Print Full Name: ---");
            for (Map<String, String> map: employees) {
                System.out.println(map.get("first_name")+" "+map.get("last_name") );
            }
        }

        }
```

TestRunner java file

```
package runner;
import io.cucumber.junit.Cucumber;
import io.cucumber.junit. CucumberOptions;
import org.junit.runner. RunWith;


@RunWith(Cucumber.class)
@CucumberOptions(

        features="src/test/resources/features",
        glue="step_definitions",
        dryRun = true,
        tags= "@listMap",

        monochrome = true,
        plugin = {"pretty","html:target/reports/cucumber-reports.html" }

)
public class TestRunner {}

// no code need inside of this class!
```

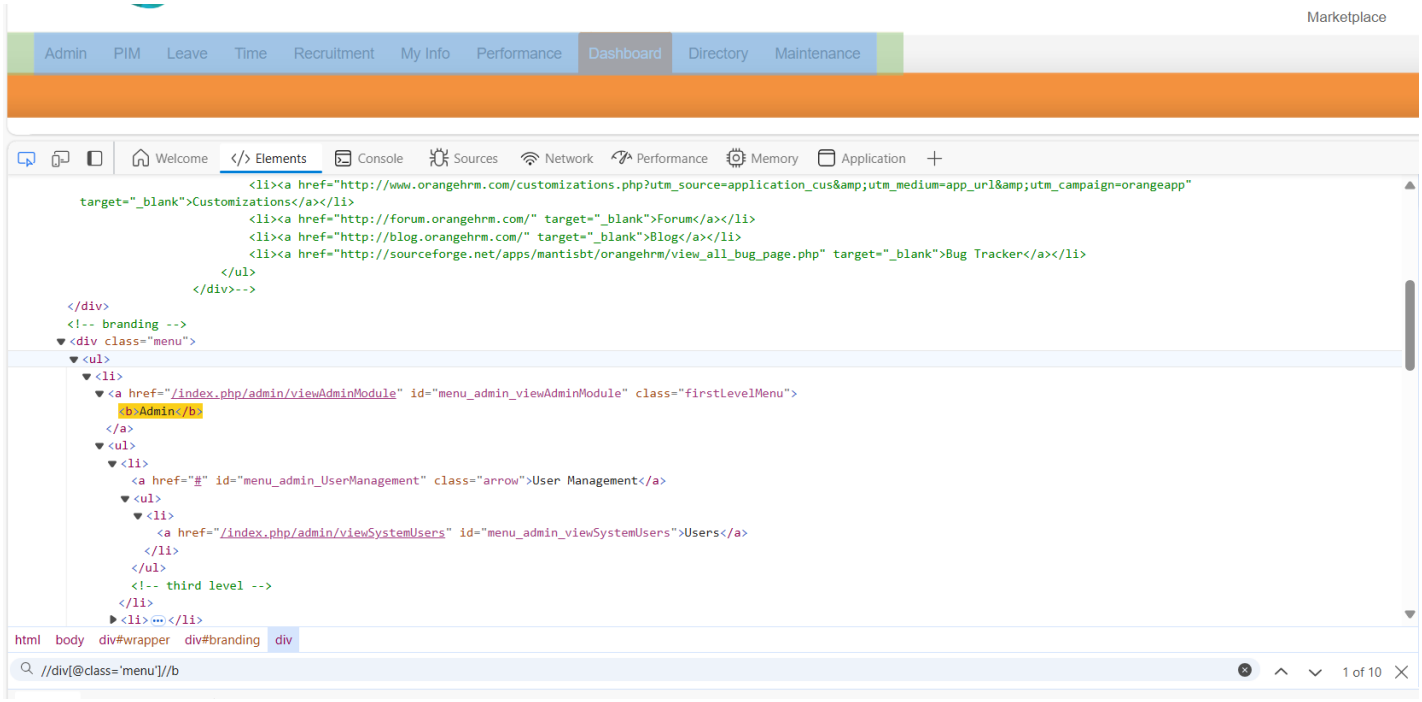
After running : TestRunner java file:


runner.TestRunner 100 ms
Demo the different ways of data tables in cucumber 100 ms
Data table as a list #1 77 ms
Data table as a list #2 0 ms
Data Table as a List of Lists 15 ms
Data Table as a List of Maps 8 ms
Example: Data table as a list # src/test/resources/features/data-table-example.feature:14
Given this is a data table as a list # step_definitions.DataTableExampleSteps.this_is_a_data_table_as_a_list(java.util.List<jav
| Omar |
| John |
| Merry |
| Jane |
@listMap @mapDataTable
Example: Data table as a list # src/test/resources/features/data-table-example.feature:21
Given this is a data table as a map # step_definitions.DataTableExampleSteps.this_is_a_data_table_as_a_map(java.util.Map<java.l
first_name	Omar
middle_name	Mark
last_name	Ramo
@listMap @listOfListsDataTable
Example: Data Table as a List of Lists # src/test/resources/features/data-table-example.feature:27
Given this is a data table as a list of Lists # step_definitions.DataTableExampleSteps.this_is_a_data_table_as_a_list_of_lists(
12313	Omar Mark	Ramo
13684	Jonathan Pete Moss	Abernathy
13724	test2 nn	Annotation
@listMap @listOfMapsDataTable
Example: Data Table as a List of Maps # src/test/resources/features/data-table-example.feature:34
Given this is a data table as a list of Maps # step_definitions.DataTableExampleSteps.this_is_a_data_table_as_a_list_of_maps(ja
id	first_name	last_name
12313	Omar Mark	Ramo
13684	Jonathan Pete Moss	Abernathy
13724	test2 nn	Annotation

Task (Need to be completed)

we need to verify the existence of (Admin, PIM, Leave, Time, Recruitment…) from the menu by using the List.

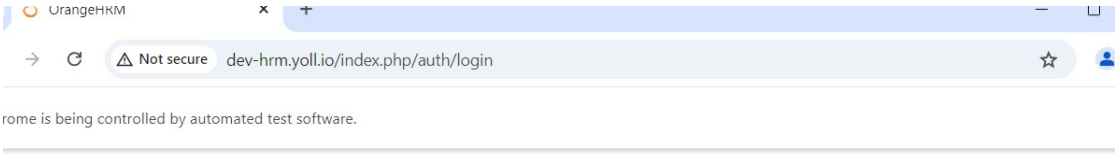
The Xpath shows the result of 10.



Task

Login to:

http://dev-hrm.yoll.io/index.php/auth/login



Feature File explaination

- **Purpose**: Defines test scenarios for login functionality, written in Gherkin syntax.
- **Background**: Ensures that the user starts on the login page for all scenarios.
- **Scenarios**:
  - **Valid Login**: Verifies login with correct credentials.
  - **Invalid Login**: Ensures error messages appear for:
    - Invalid credentials.
    - Empty password.
    - Empty username.
    - Completely empty credentials.

- Each scenario includes clear **steps** (Given, When, Then) for action and verification

@regression

```
Feature: This feature file validates Login Page Functionality
Background:
  Given user navigates to HRM Login Page
  @regression
  Scenario: User should be able to login to HRM with valid credentials
    #Given user navigates to HRM Login Page
    When user enters "yoll-student" username and "Bootcamp5#" password and clicks on
login button
      #When user enters "yoll-student" username
    #Then user enters "Bootcamp5#" password
    #Then user clicks on login button
    And user should be able to verify account name header "Welcome Yoll" on HRM Homepage


  Scenario: User should not be able to login to HRM with invalid credentials
    #Given user navigates to HRM Login Page
    When user enters "xyz" username and "abc#" password and clicks on login button
    #When user enters "xyz" username
    #Then user enters "abc#" password
   # When user enters invalid username
   # Then user enters invalid password
    #Then user clicks on login button
    And user validates "Invalid credentials" error message

  Scenario: User should not be able to login to HRM with empty password
   # Given user navigates to HRM Login Page
    When user enters "xyz" username and "" password and clicks on login button
   # When user enters "xyz" username
    #Then user enters "" password
   # When user enters invalid username
   # Then user enters empty password
    #Then user clicks on login button
    And user validates "Password cannot be empty" error message

  Scenario: User should not be able to login to HRM with empty username
   # Given user navigates to HRM Login Page
    When user enters "" username and "abc" password and clicks on login button
    #When user enters "" username
    #Then user enters "abc" password
    #When user enters empty username
   # Then user enters invalid password
    #Then user clicks on login button
    And user validates "Username cannot be empty" error message

  Scenario: User should not be able to login to HRM with empty credentials
    #Given user navigates to HRM Login Page
    When user enters "" username and "" password and clicks on login button
    #When user enters "" username
    #Then user enters "" password
```

```
    #When user enters empty username
    #Then user enters empty password
    #Then user clicks on login button
    And user validates "Username cannot be empty" error message
```

HomePage

```
package pages;

    import org.junit.Assert;
    import org.openqa.selenium.WebDriver;
    import org.openqa.selenium.WebElement;
    import org.openqa.selenium.support.FindBy;
    import org.openqa.selenium.support.PageFactory;

    public class HomePage {
        @FindBy(id = "welcome")
        //@FindBy(xpath = "//*[@id='welcome']")
        // @FindBy(css = "#welcome")
        public WebElement userNameHeader;
        @FindBy(id = "menu_admin_viewAdminModule")
        WebElement adminTab;
        @FindBy(id = "menu_pim_viewPimModule")
        WebElement pimTab;
        @FindBy(id = "menu_admin_Job")
        WebElement jobSubTab;
        @FindBy(id = "menu_pim_viewEmployeeList")
        WebElement employeeListSubTab;
        @FindBy(id = "menu_admin_viewJobTitleList")
        WebElement jobTitlesOption;
        public HomePage(WebDriver driver) {
            PageFactory.initElements(driver, this);
        }

        public void validateUsernameHeader(String expectedUsernameHeader) {
            String actualUsernameHeader = userNameHeader.getText();
            Assert.assertEquals(actualUsernameHeader, expectedUsernameHeader, "Username
header verification failed!");
        }

        public void navigateToJobTitlesPage() throws InterruptedException {
            adminTab.click();
            Thread.sleep(1000);
            jobSubTab.click();
            Thread.sleep(1000);
            jobTitlesOption.click();
            Thread.sleep(1000);
        }

        public void navigateToEmployeeListSubTab() throws InterruptedException {
            pimTab.click();
            Thread.sleep(1000);
            employeeListSubTab.click();
            Thread.sleep(1000);
        }

    }
```

LoginPage

```
    package pages;

    import org.junit.Assert;
    import org.openqa.selenium.WebDriver;
    import org.openqa.selenium.WebElement;
    import org.openqa.selenium.support.FindBy;
    import org.openqa.selenium.support.PageFactory;

    public class LoginPage {

        public LoginPage (WebDriver driver) {

    PageFactory.initElements (driver, this);}
        @FindBy(id="txtUsername")
        public WebElement usernameTextInput;
        @FindBy(id="txtPassword")
        public WebElement passwordTextInput;
        @FindBy(xpath="//*[@id='btnLogin']")
        public WebElement loginButton;
        @FindBy(id = "spanMessage")
        public WebElement errorMessage;
        public void login (String username, String password) {
            usernameTextInput.sendKeys(username);
            passwordTextInput.sendKeys(password);
            loginButton.click();
        }
        public void verifyErrorMessage (String expectedErrorMessage) {

        String actualErrorMessage = errorMessage.getText();
    Assert.assertEquals("Error message validation
failed!",actualErrorMessage,expectedErrorMessage);
    }}
```

TestRunner

```
    package runner;

    import io.cucumber.junit. Cucumber;
    import io.cucumber.junit.CucumberOptions;
    import org.junit.runner. RunWith;
    @RunWith(Cucumber.class)
    @CucumberOptions(
            features="src/test/resources/Features",
            glue = {"step_definitions"},
            dryRun= false ,
            tags= "@regression",
            plugin={"pretty", "html:target/cucumber-reports"},
            monochrome = false
    )

    public class TestRunner {
    }
```

HomePageSteps

```
    package step_definitions;
```

```
    import io.cucumber.java.en.And;
    import io.cucumber.java.en.Then;
    import org.junit.Assert;
    import org.openqa.selenium.By;
    import org.openqa.selenium.WebDriver;
    import org.openqa.selenium.support.PageFactory;
    import pages.HomePage;
    import utilities.DriverFactory;

    public class HomePageSteps {
        WebDriver driver =DriverFactory.getDriver();
        private HomePage homePage = new HomePage(driver);

//        @Then("user should be able to verify account name header on HRM Homepage")
//        public void
user_should_be_able_to_verify_account_name_header_on_hrm_homepage() {
//            String expectedHeader = "Welcome Yoll";
//            String actualHeader = driver.findElement(By.id("welcome")).getText();
//            Assert.assertEquals("Account name header verification failed!",
expectedHeader, actualHeader);
//        }

        @And("user should be able to verify account name header {string} on HRM
Homepage")
        public void userShouldBeAbleToVerifyAccountNameHeaderOnHRMHomepage(String
userNameHeader) {
        }
    }
```

LoginPageSteps

```
    package step_definitions;

    import io.cucumber.java.en.Given;
    import io.cucumber.java.en.Then;
    import io.cucumber.java.en.When;
    import org.junit.Assert;
    import org.openqa.selenium.By;
    import org.openqa.selenium.WebDriver;

    import pages.LoginPage;
    import utilities.DriverFactory;

    public class LoginPageSteps {

        WebDriver driver = DriverFactory.getDriver();
        LoginPage loginPage = new LoginPage(driver);

        @Given("user navigates to HRM Login Page")
        public void user_navigates_to_hrm_login_page() {
            driver.get("http://dev-hrm.yoll.io/index.php/auth/login");
        }
    //=========================
//        @When("user enters username")
//        public void user_enters_username() {
//            loginPage.usernameTextInput.sendKeys("yoll-student");
    //
//        }
    //
//        @Then("user enters password")
```

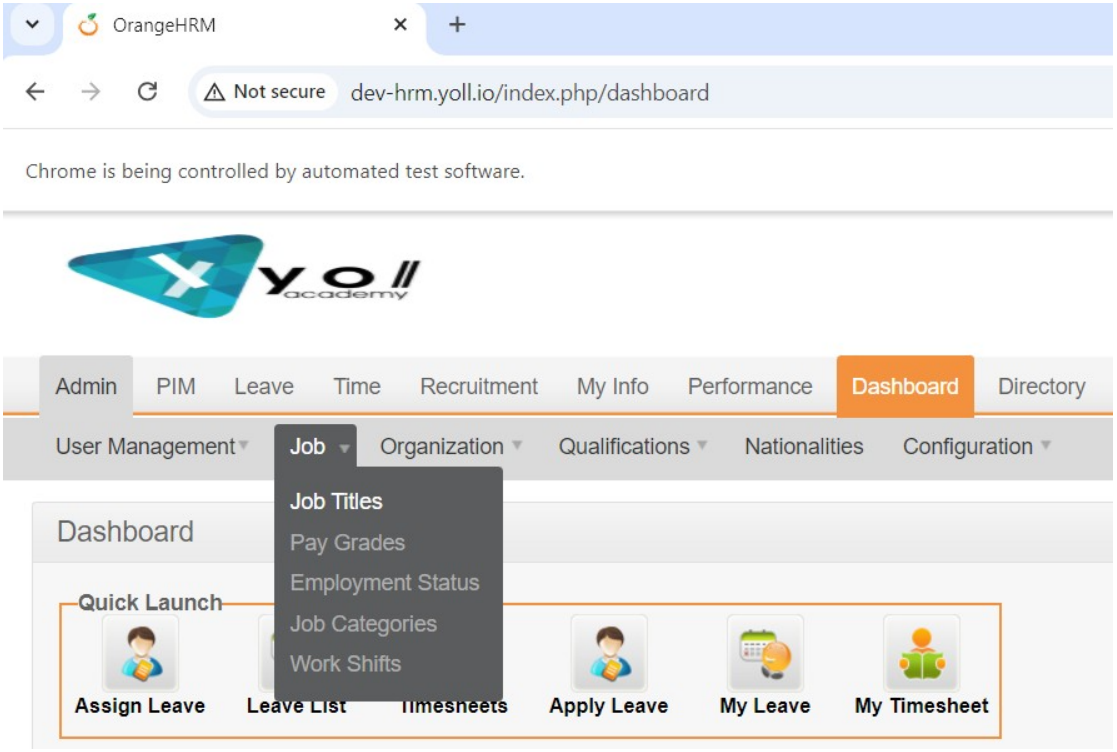
```
//        public void user_enters_password() {
//            loginPage.passwordTextInput.sendKeys("Bootcamp#5");
    //
//        }

//        @When("user enters {string} username")
//        public void user_enters_username(String username) {
//            loginPage.usernameTextInput.sendKeys("yoll-student");
    //
//        }
//        @Then("user enters {string} password")
//        public void user_enters_password(String password) {
//            loginPage.passwordTextInput.sendKeys("Bootcamp#5");
    //
//        }
    //
//        //=========================
    //
//        @Then("user clicks on login button")
//        public void user_clicks_on_login_button() {
//            loginPage.loginButton.click();
//        }
        @When("user enters {string} username and {string} password and clicks on login
button")
        public void user_enters_username_and_password_and_clicks_on_login_button(String
username, String password) {
            loginPage.login(username,password);
        }
        @Then("user validates {string} error message")
        public void user_validates_error_message(String errorMessage) {
           loginPage.verifyErrorMessage(errorMessage);
        }

//        @Then("user should be able to verify account name header on HRM Homepage")
//        public void
user_should_be_able_to_verify_account_name_header_on_hrm_homepage() {
//            // Write code here that turns the phrase above into concrete actions
    //
//            String expectedHeader = "Welcome Yoll";
//            String actualHeader = driver.findElement(By.id("welcome")).getText();
//            Assert.assertEquals("Account name header verification failed!",
expectedHeader, actualHeader);
//        }
        }
```

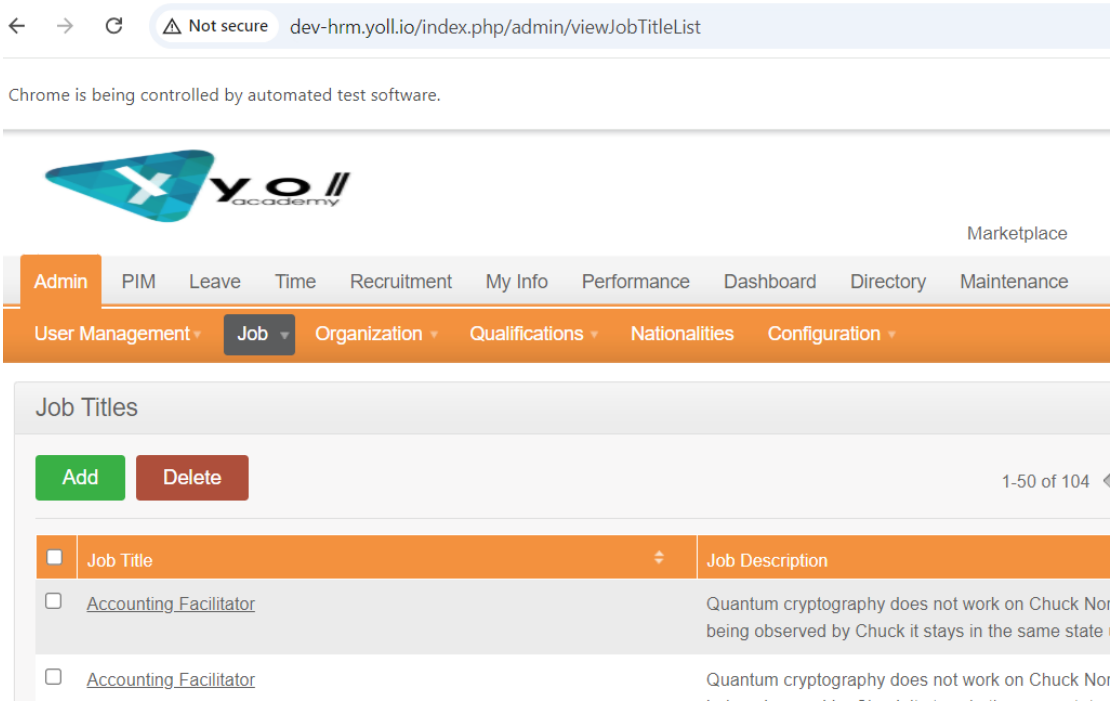
DriverFactory

```
    package utilities;
    import io.github.bonigarcia.wdm.WebDriverManager;
    import org.openqa.selenium. WebDriver;
    import org.openqa.selenium.chrome.ChromeDriver;
    public class DriverFactory {
        private static WebDriver driver;
        public static WebDriver getDriver () {
            if (driver == null) {
                WebDriverManager.chromedriver().setup();
                driver = new ChromeDriver();
            }
            return driver;
        }
```

```
        public static void closeDriver () {
            if (driver != null) {
                driver.close();
                driver = null;
            }
        }
    }
```

Task






```
@jobTitles
Feature: This feature file validates Job Titles Page Functionality
  Background:
    Given user navigates to HRM Login Page
    When user enters "yoll-student" username and "Bootcamp5#" password and
clicks on login button
    And user should be able to verify account name header "Welcome Yoll" on HRM
Homepage
    Then user navigates to "Job Titles" Page

  Scenario: User should be able to navigate to Job Titles Page on HRM
    And user validates Job Titles page header

  Scenario: User should be able to validate buttons on Job Titles Page on HRM
    And user validates buttons on Job Titles Page

  Scenario: User should be able to add new job title on Job Titles Page on HRM
    And user creates a new job title on Job Titles Page
```

```
package pages;

import org.junit.Assert;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.FindBy;
import org.openqa.selenium.support.PageFactory;

public class HomePage {
    @FindBy(id = "welcome")
    //@FindBy(xpath = "//*[@id='welcome']")
    // @FindBy(css = "#welcome")
    public WebElement userNameHeader;
    @FindBy(id = "menu_admin_viewAdminModule")
```

```
    WebElement adminTab;
    @FindBy(id = "menu_pim_viewPimModule")
    WebElement pimTab;
    @FindBy(id = "menu_admin_Job")
    WebElement jobSubTab;
    @FindBy(id = "menu_pim_viewEmployeeList")
    WebElement employeeListSubTab;
    @FindBy(id = "menu_admin_viewJobTitleList")
    WebElement jobTitlesOption;
    public HomePage(WebDriver driver) {
        PageFactory.initElements(driver, this);
    }
    public void verifyUsernameHeader (String expectedUsernameHeader) {
        String actualUsernameHeader = userNameHeader.getText();
        Assert.assertEquals("Username header verification failed!",
                expectedUsernameHeader, actualUsernameHeader);
    }

    public void validateUsernameHeader(String expectedUsernameHeader) {
        String actualUsernameHeader = userNameHeader.getText();
        Assert.assertEquals(actualUsernameHeader, expectedUsernameHeader,
"Username header verification failed!");
    }

    public void navigateToJobTitlesPage() throws InterruptedException {
        adminTab.click();
        Thread.sleep(1000);
        jobSubTab.click();
        Thread.sleep(1000);
        jobTitlesOption.click();
        Thread.sleep(1000);
    }

    public void navigateToEmployeeListSubTab() throws InterruptedException {
        pimTab.click();
        Thread.sleep(1000);
        employeeListSubTab.click();
        Thread.sleep(1000);
    }

}
```

```
package pages;
import org.junit.Assert;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.FindBy;
import org.openqa.selenium.support.PageFactory;
import java.util.List;
public class JobTitlesPage {
    public JobTitlesPage (WebDriver driver) {
```

```
        PageFactory.initElements(driver, this);
    }
    @FindBy(xpath = "//*[@class='head']/h1")
    WebElement jobTitlesHeader;
    @FindBy(id = "btnAdd")
    WebElement addButton;
    @FindBy(id = "btnDelete")
    WebElement deleteButton;
    @FindBy(id = "jobTitle_jobTitle")
    WebElement jobTitleTextBox;
    @FindBy(id = "jobTitle_jobDescription")
    WebElement jobDescriptionTextBox;
    @FindBy(id = "btnSave")
    WebElement saveButton;
    @FindBy(id = "dialogDeleteBtn")
    WebElement modalWindowOKButton;
    @FindBy(xpath = "//table[@id='resultTable']/tbody/tr/td[2]/a")
    List<WebElement> allJobTitles;
    public void validatePageHeader () {
        String expectedPageHeader = "Job Titles";
        String actualPageHeader = jobTitlesHeader.getText();
        Assert.assertEquals("Page title verification failed!",
                expectedPageHeader, actualPageHeader);
    }
    public void verifyButtonsDisplayed () throws InterruptedException {
        Thread.sleep(2000);
        Assert.assertTrue("Add button verification failed!",
addButton.isDisplayed());
        Assert.assertTrue("Delete button verification failed!",
deleteButton.isDisplayed());
    }
    public void addJobTitle (String jobTitle, String jobDescription) throws
InterruptedException {
        Thread.sleep(2000);
        addButton.click();
        Thread.sleep(2000);
        jobTitleTextBox.sendKeys(jobTitle);
        Thread.sleep(1000);
        jobDescriptionTextBox.sendKeys(jobDescription);
        Thread.sleep(1000);
        saveButton.click();
    }
    public boolean verifyNewJobTitle (String expectedJobTitle) throws
InterruptedException {
        Thread.sleep(2000);
        boolean isNewJobTitleCreated = false;
        for (WebElement jobTitle : allJobTitles) {
            String actualJobTitleValue = jobTitle.getText();
            if (actualJobTitleValue.equals(expectedJobTitle)) {
                isNewJobTitleCreated = true;
                break;
            }
        }
        return isNewJobTitleCreated;
    }
```

```
    public boolean verifyJobTitleDeleted (WebDriver driver, String
expectedJobTitle) throws InterruptedException {
        Thread.sleep(2000);
        boolean isJobTitleDeleted = false;
        for (WebElement jobTitle : allJobTitles) {
            String actualJobTitleValue = jobTitle.getText();
            if (actualJobTitleValue.equals(expectedJobTitle)) {

driver.findElement(By.xpath("//table[@id='resultTable']/tbody/tr/td[2]/
a[text()='"+expectedJobTitle+"']/../../td[1]")).click();
                Thread.sleep(1000);
                deleteButton.click();
                Thread.sleep(1000);
                modalWindowOKButton.click();
                isJobTitleDeleted = true;
                break;
            }
        }
        return isJobTitleDeleted;
    }
}


package pages;

import org.junit.Assert;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.FindBy;
import org.openqa.selenium.support.PageFactory;

public class LoginPage {

    public LoginPage (WebDriver driver) {

PageFactory.initElements (driver, this);}
    @FindBy(id="txtUsername")
    public WebElement usernameTextInput;
    @FindBy(id="txtPassword")
    public WebElement passwordTextInput;
    @FindBy(xpath="//*[@id='btnLogin']")
    public WebElement loginButton;
    @FindBy(id = "spanMessage")
    public WebElement errorMessage;
    public void login (String username, String password) {
        usernameTextInput.sendKeys(username);
        passwordTextInput.sendKeys(password);
        loginButton.click();
    }
    public void verifyErrorMessage (String expectedErrorMessage) {

    String actualErrorMessage = errorMessage.getText();
Assert.assertEquals("Error message validation
failed!",actualErrorMessage,expectedErrorMessage);
}}
```

```java
package runner;

import io.cucumber.junit. Cucumber;
import io.cucumber.junit.CucumberOptions;
import org.junit.runner. RunWith;
@RunWith(Cucumber.class)
@CucumberOptions(
        features="src/test/resources/Features/jobTitles.feature",
        glue = {"step_definitions"},
        dryRun= true ,
        tags= "@jobTitles",
        plugin={"pretty", "html:target/cucumber-reports"},
        monochrome = false
)

public class TestRunner {
}


package step_definitions;

import io.cucumber.java.en.And;
import io.cucumber.java.en.Then;
import org.junit.Assert;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.support.PageFactory;
import pages.HomePage;
import utilities.DriverFactory;

public class HomePageSteps {
    WebDriver driver =DriverFactory.getDriver();
    private HomePage homePage = new HomePage(driver);

//    @Then("user should be able to verify account name header on HRM Homepage")
//    public void
user_should_be_able_to_verify_account_name_header_on_hrm_homepage() {
//        String expectedHeader = "Welcome Yoll";
//        String actualHeader = driver.findElement(By.id("welcome")).getText();
//        Assert.assertEquals("Account name header verification failed!",
expectedHeader, actualHeader);
//    }

    @And("user should be able to verify account name header {string} on HRM
Homepage")
    public void userShouldBeAbleToVerifyAccountNameHeaderOnHRMHomepage(String
usernameHeader) {
        homePage.verifyUsernameHeader(usernameHeader);
    }
    @Then("user navigates to {string} Page")
    public void user_navigates_to_Page(String page) throws InterruptedException
{
        switch (page) {
```

```
            case "Job Titles":
                homePage.navigateToJobTitlesPage();
                break;
            case "Employee List":
                homePage.navigateToEmployeeListSubTab();
                break;
        }
    }
    @Then("user validates {string} page header")
    public void user_validates_page_header(String pageHeader) {
    }

}

package step_definitions;
import pages.JobTitlesPage;
import utilities.DriverFactory;
import io.cucumber.java.en.And;
import io.cucumber.java.en.Then;
import org.openqa.selenium.WebDriver;
public class JobTitlesPageSteps {
    WebDriver driver = DriverFactory.getDriver();
    JobTitlesPage jobTitlesPage = new JobTitlesPage(driver);
    @Then("user validates Job Titles page header")
    public void user_validates_page_header() {
        jobTitlesPage.validatePageHeader();
    }
    @And("user validates buttons on Job Titles Page")
    public void userValidatesButtonsOnJobTitlesPage() throws
InterruptedException {
        jobTitlesPage.verifyButtonsDisplayed();
    }
    @Then("user creates a new job title on Job Titles Page")
    public void user_creates_a_new_job_title_on_job_titles_page() throws
InterruptedException {
        String jobTitle = "JobTitle001";
        String jobDescription = "Automation Test Engineer";

        jobTitlesPage.verifyJobTitleDeleted(driver,jobTitle);
        jobTitlesPage.addJobTitle(jobTitle,jobDescription);
        jobTitlesPage.verifyNewJobTitle(jobTitle);
    }
}


package step_definitions;

import io.cucumber.java.en.Given;
import io.cucumber.java.en.Then;
import io.cucumber.java.en.When;
import org.junit.Assert;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;

import pages.LoginPage;
```

```
import utilities.DriverFactory;

public class LoginPageSteps {

    WebDriver driver = DriverFactory.getDriver();
    LoginPage loginPage = new LoginPage(driver);

    @Given("user navigates to HRM Login Page")
    public void user_navigates_to_hrm_login_page() {
        driver.get("http://dev-hrm.yoll.io/index.php/auth/login");
    }
//=========================
//    @When("user enters username")
//    public void user_enters_username() {
//        loginPage.usernameTextInput.sendKeys("yoll-student");
//
//    }
//
//    @Then("user enters password")
//    public void user_enters_password() {
//        loginPage.passwordTextInput.sendKeys("Bootcamp#5");
//
//    }

//    @When("user enters {string} username")
//    public void user_enters_username(String username) {
//        loginPage.usernameTextInput.sendKeys("yoll-student");
//
//    }
//    @Then("user enters {string} password")
//    public void user_enters_password(String password) {
//        loginPage.passwordTextInput.sendKeys("Bootcamp#5");
//
//    }
//
//    //=========================
//
//    @Then("user clicks on login button")
//    public void user_clicks_on_login_button() {
//        loginPage.loginButton.click();
//    }
    @When("user enters {string} username and {string} password and clicks on
login button")
    public void
user_enters_username_and_password_and_clicks_on_login_button(String username,
String password) {
        loginPage.login(username,password);
    }
    @Then("user validates {string} error message")
    public void user_validates_error_message(String errorMessage) {
       loginPage.verifyErrorMessage(errorMessage);
    }

//    @Then("user should be able to verify account name header on HRM Homepage")
```

```
//    public void
user_should_be_able_to_verify_account_name_header_on_hrm_homepage() {
//        // Write code here that turns the phrase above into concrete actions
//
//        String expectedHeader = "Welcome Yoll";
//        String actualHeader = driver.findElement(By.id("welcome")).getText();
//        Assert.assertEquals("Account name header verification failed!",
expectedHeader, actualHeader);
//    }
    }
```

```
package utilities;
import io.github.bonigarcia.wdm.WebDriverManager;
import org.openqa.selenium. WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
public class DriverFactory {
    private static WebDriver driver;
    public static WebDriver getDriver () {
        if (driver == null) {
            WebDriverManager.chromedriver().setup();
            driver = new ChromeDriver();
        }
        return driver;
    }
        public static void closeDriver () {
            if (driver != null) {
                driver.close();
                driver = null;
            }
        }
    }
```

Task : SCENARIO OUTLINE

loginPage.feature

```
@regression
Feature: This feature file validates Login Page Functionality
Background:
  Given user navigates to HRM Login Page

  Scenario: User should be able to login to HRM with valid credentials
    #Given user navigates to HRM Login Page
    When user enters "yoll-student" username and "Bootcamp5#" password and
clicks on login button
      #When user enters "yoll-student" username
    #Then user enters "Bootcamp5#" password
    #Then user clicks on login button
```

```
    And user should be able to verify account name header "Welcome Yoll" on HRM
Homepage

  Scenario Outline: User should not be able to login to HRM with invalid
credentials
    When user enters "<username>" username and "<password>" password and clicks
on login button
    And user validates "<errorMessage>" error message
    Examples:
      | username | password | errorMessage             |
      | xyz      | abc      | Invalid credentials      |
      | xyz      |          | Password cannot be empty |
      |          | abc      | Username cannot be empty |
      |          |          | Username cannot be empty |

  #Scenario: User should not be able to login to HRM with invalid credentials
    #Given user navigates to HRM Login Page
   ## And user validates "Invalid credentials" error message

  #Scenario: User should not be able to login to HRM with empty password
   # Given user navigates to HRM Login Page
   # When user enters "xyz" username and "" password and clicks on login button
   # When user enters "xyz" username
    #Then user enters "" password
   # When user enters invalid username
   # Then user enters empty password
    #Then user clicks on login button
    #And user validates "Password cannot be empty" error message

 # Scenario: User should not be able to login to HRM with empty username
   # Given user navigates to HRM Login Page
   # When user enters "" username and "abc" password and clicks on login button
    #When user enters "" username
    #Then user enters "abc" password
    #When user enters empty username
   # Then user enters invalid password
    #Then user clicks on login button
   # And user validates "Username cannot be empty" error message

 # Scenario: User should not be able to login to HRM with empty credentials
    #Given user navigates to HRM Login Page
   # When user enters "" username and "" password and clicks on login button
    #When user enters "" username
    #Then user enters "" password
    #When user enters empty username
    #Then user enters empty password
    #Then user clicks on login button
    #And user validates "Username cannot be empty" error message
```

jobTitles.feature

```
@jobTitles
Feature: This feature file validates Job Titles Page Functionality
  Background:
    Given user navigates to HRM Login Page
```

```
    When user enters "yoll-student" username and "Bootcamp5#" password and
clicks on login button
    And user should be able to verify account name header "Welcome Yoll" on HRM
Homepage
    Then user navigates to "Job Titles" Page

  Scenario: User should be able to navigate to Job Titles Page on HRM
    And user validates Job Titles page header

  Scenario: User should be able to validate buttons on Job Titles Page on HRM
    And user validates buttons on Job Titles Page

  Scenario: User should be able to add new job title on Job Titles Page on HRM
    And user creates a new job title on Job Titles Page
```

HomePage.feature

```
@regression @homePage
Feature: This feature file validates Home Page Functionality
  Background:
    Given user navigates to HRM Login Page
    When user enters "yoll-student" username and "Bootcamp5#" password and
clicks on login button
    And user should be able to verify account name header "Welcome Yoll" on HRM
Homepage
  Scenario: User should be able to validate main tabs on HRM Homepage
    And user validates main tabs on HRM Homepage
      | tabname |
      | Admin |
      | PIM |
      | Leave |
      | Time |
      | Recruitment |
      | My Info |
      | Performance |
      | Dashboard |
      | Directory |
      | Maintenance |
```

```
package pages;

import org.junit.Assert;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.FindBy;
import org.openqa.selenium.support.PageFactory;

import java.util.List;

public class HomePage {
    @FindBy(id = "welcome")
    //@FindBy(xpath = "//*[@id='welcome']")
    // @FindBy(css = "#welcome")
    public WebElement userNameHeader;
```

```
    @FindBy(id = "menu_admin_viewAdminModule")
    WebElement adminTab;
    @FindBy(id = "menu_pim_viewPimModule")
    WebElement pimTab;
    @FindBy(id = "menu_admin_Job")
    WebElement jobSubTab;
    @FindBy(id = "menu_pim_viewEmployeeList")
    WebElement employeeListSubTab;
    @FindBy(id = "menu_admin_viewJobTitleList")
    WebElement jobTitlesOption;
    @FindBy(xpath = "//*[@class='menu']/ul/li")
    public List<WebElement> allMainTabs;

    public HomePage(WebDriver driver) {
        PageFactory.initElements(driver, this);
    }
    public void verifyUsernameHeader (String expectedUsernameHeader) {
        String actualUsernameHeader = userNameHeader.getText();
        Assert.assertEquals("Username header verification failed!",
                expectedUsernameHeader, actualUsernameHeader);
    }

    public void validateUsernameHeader(String expectedUsernameHeader) {
        String actualUsernameHeader = userNameHeader.getText();
        Assert.assertEquals(actualUsernameHeader, expectedUsernameHeader,
"Username header verification failed!");
    }

    public void navigateToJobTitlesPage() throws InterruptedException {
        adminTab.click();
        Thread.sleep(1000);
        jobSubTab.click();
        Thread.sleep(1000);
        jobTitlesOption.click();
        Thread.sleep(1000);
    }

    public void navigateToEmployeeListSubTab() throws
InterruptedException {
        pimTab.click();
        Thread.sleep(1000);
        employeeListSubTab.click();
        Thread.sleep(1000);
    }

}



package pages;
```

```
import org.junit.Assert;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.FindBy;
import org.openqa.selenium.support.PageFactory;
import java.util.List;
public class JobTitlesPage {
    public JobTitlesPage (WebDriver driver) {
        PageFactory.initElements(driver, this);
    }
    @FindBy(xpath = "//*[@class='head']/h1")
    WebElement jobTitlesHeader;
    @FindBy(id = "btnAdd")
    WebElement addButton;
    @FindBy(id = "btnDelete")
    WebElement deleteButton;
    @FindBy(id = "jobTitle_jobTitle")
    WebElement jobTitleTextBox;
    @FindBy(id = "jobTitle_jobDescription")
    WebElement jobDescriptionTextBox;
    @FindBy(id = "btnSave")
    WebElement saveButton;
    @FindBy(id = "dialogDeleteBtn")
    WebElement modalWindowOKButton;
    @FindBy(xpath = "//table[@id='resultTable']/tbody/tr/td[2]/a")
    List<WebElement> allJobTitles;
    public void validatePageHeader () {
        String expectedPageHeader = "Job Titles";
        String actualPageHeader = jobTitlesHeader.getText();
        Assert.assertEquals("Page title verification failed!",
                expectedPageHeader, actualPageHeader);
    }
    public void verifyButtonsDisplayed () throws InterruptedException {
        Thread.sleep(2000);
        Assert.assertTrue("Add button verification failed!",
addButton.isDisplayed());
        Assert.assertTrue("Delete button verification failed!",
deleteButton.isDisplayed());
    }
    public void addJobTitle (String jobTitle, String jobDescription)
throws InterruptedException {
        Thread.sleep(2000);
        addButton.click();
        Thread.sleep(2000);
        jobTitleTextBox.sendKeys(jobTitle);
        Thread.sleep(1000);
        jobDescriptionTextBox.sendKeys(jobDescription);
        Thread.sleep(1000);
        saveButton.click();
    }
```

```
    public boolean verifyNewJobTitle (String expectedJobTitle) throws
InterruptedException {
        Thread.sleep(2000);
        boolean isNewJobTitleCreated = false;
        for (WebElement jobTitle : allJobTitles) {
            String actualJobTitleValue = jobTitle.getText();
            if (actualJobTitleValue.equals(expectedJobTitle)) {
                isNewJobTitleCreated = true;
                break;
            }
        }
        return isNewJobTitleCreated;
    }
    public boolean verifyJobTitleDeleted (WebDriver driver, String
expectedJobTitle) throws InterruptedException {
        Thread.sleep(2000);
        boolean isJobTitleDeleted = false;
        for (WebElement jobTitle : allJobTitles) {
            String actualJobTitleValue = jobTitle.getText();
            if (actualJobTitleValue.equals(expectedJobTitle)) {

driver.findElement(By.xpath("//table[@id='resultTable']/tbody/tr/td[2]/
a[text()='"+expectedJobTitle+"']/../../td[1]")).click();
                Thread.sleep(1000);
                deleteButton.click();
                Thread.sleep(1000);
                modalWindowOKButton.click();
                isJobTitleDeleted = true;
                break;
            }
        }
        return isJobTitleDeleted;
    }
}


package pages;

import org.junit.Assert;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.FindBy;
import org.openqa.selenium.support.PageFactory;

public class LoginPage {

    public LoginPage (WebDriver driver) {

PageFactory.initElements (driver, this);}
    @FindBy(id="txtUsername")
    public WebElement usernameTextInput;
```

```
    @FindBy(id="txtPassword")
    public WebElement passwordTextInput;
    @FindBy(xpath="//*[@id='btnLogin']")
    public WebElement loginButton;
    @FindBy(id = "spanMessage")
    public WebElement errorMessage;
    public void login (String username, String password) {
        usernameTextInput.sendKeys(username);
        passwordTextInput.sendKeys(password);
        loginButton.click();
    }
    public void verifyErrorMessage (String expectedErrorMessage) {

    String actualErrorMessage = errorMessage.getText();
Assert.assertEquals("Error message validation
failed!",actualErrorMessage,expectedErrorMessage);
}}
```

```
package runner;

import io.cucumber.junit. Cucumber;
import io.cucumber.junit.CucumberOptions;
import org.junit.runner. RunWith;
@RunWith(Cucumber.class)
@CucumberOptions(
        features="src/test/resources/Features/loginPage.feature",
        glue = {"step_definitions"},
        dryRun= false ,
        tags= "@regression",
        plugin={"pretty", "html:target/cucumber-reports"},
        monochrome = false
)

public class TestRunner {
}
```

```
package step_definitions;

import io.cucumber.datatable.DataTable;
import io.cucumber.java.en.And;
import io.cucumber.java.en.Then;
import org.junit.Assert;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.support.PageFactory;
import pages.HomePage;
import utilities.DriverFactory;
```

```java
import java.util.List;
import java.util.Map;

public class HomePageSteps {
    WebDriver driver = DriverFactory.getDriver();
    private HomePage homePage = new HomePage(driver);

//    @Then("user should be able to verify account name header on HRM
Homepage")
//    public void
user_should_be_able_to_verify_account_name_header_on_hrm_homepage() {
//        String expectedHeader = "Welcome Yoll";
//        String actualHeader =
driver.findElement(By.id("welcome")).getText();
//        Assert.assertEquals("Account name header verification failed!",
expectedHeader, actualHeader);
//    }

    @And("user should be able to verify account name header {string} on
HRM Homepage")
    public void
userShouldBeAbleToVerifyAccountNameHeaderOnHRMHomepage(String
usernameHeader) {
        homePage.verifyUsernameHeader(usernameHeader);
    }

    @Then("user navigates to {string} Page")
    public void user_navigates_to_Page(String page) throws
InterruptedException {
        switch (page) {
            case "Job Titles":
                homePage.navigateToJobTitlesPage();
                break;
            case "Employee List":
                homePage.navigateToEmployeeListSubTab();
                break;
        }
    }

    @Then("user validates {string} page header")
    public void user_validates_page_header(String pageHeader) {
    }

    @And("user validates main tabs on HRM Homepage")
    public void userValidatesMainTabsOnHRMHomepage(DataTable dataTable) {

        List<Map<String, String>> maps = dataTable.asMaps(String.class,
String.class);
        int i=0;
        for (Map<String, String> map:maps) {
```

```
            String expectedTab = map.get("tabname");
            String actualTab = homePage.allMainTabs.get(i).getText();
            Assert.assertEquals( "Tab name verification failed!",
expectedTab, actualTab);
            i++;
        }
    }
}
```

```
package step_definitions;
import pages.JobTitlesPage;
import utilities.DriverFactory;
import io.cucumber.java.en.And;
import io.cucumber.java.en.Then;
import org.openqa.selenium.WebDriver;
public class JobTitlesPageSteps {
    WebDriver driver = DriverFactory.getDriver();
    JobTitlesPage jobTitlesPage = new JobTitlesPage(driver);
    @Then("user validates Job Titles page header")
    public void user_validates_page_header() {
        jobTitlesPage.validatePageHeader();
    }
    @And("user validates buttons on Job Titles Page")
    public void userValidatesButtonsOnJobTitlesPage() throws
InterruptedException {
        jobTitlesPage.verifyButtonsDisplayed();
    }
    @Then("user creates a new job title on Job Titles Page")
    public void user_creates_a_new_job_title_on_job_titles_page() throws
InterruptedException {
        String jobTitle = "JobTitle001";
        String jobDescription = "Automation Test Engineer";

        jobTitlesPage.verifyJobTitleDeleted(driver,jobTitle);
        jobTitlesPage.addJobTitle(jobTitle,jobDescription);
        jobTitlesPage.verifyNewJobTitle(jobTitle);
    }
}
```

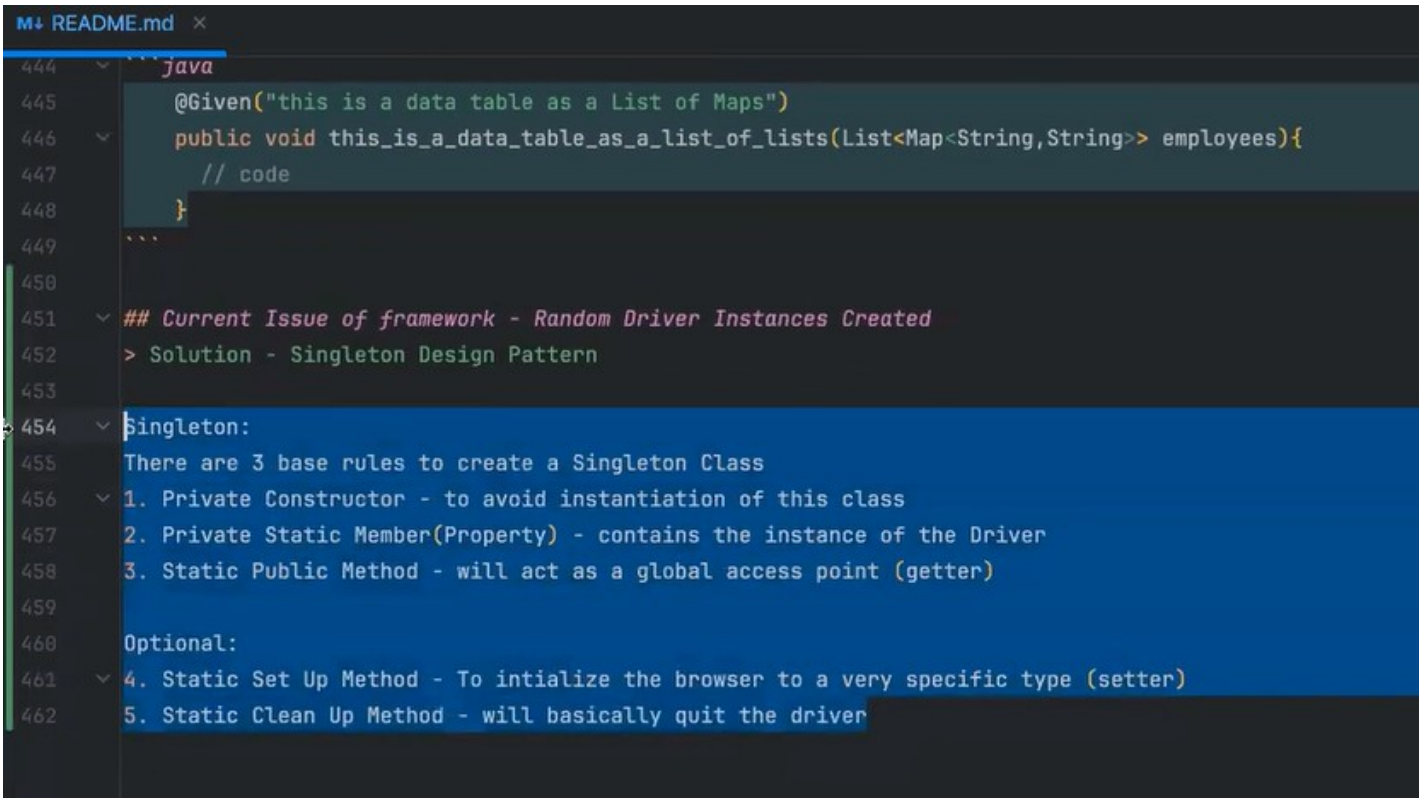
```
package step_definitions;

import io.cucumber.java.en.Given;
import io.cucumber.java.en.Then;
import io.cucumber.java.en.When;
import org.junit.Assert;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;

import pages.LoginPage;
import utilities.DriverFactory;
```

```
public class LoginPageSteps {

    WebDriver driver = DriverFactory.getDriver();
    LoginPage loginPage = new LoginPage(driver);

    @Given("user navigates to HRM Login Page")
    public void user_navigates_to_hrm_login_page() {
        driver.get("http://dev-hrm.yoll.io/index.php/auth/login");
    }
//=========================
//    @When("user enters username")
//    public void user_enters_username() {
//        loginPage.usernameTextInput.sendKeys("yoll-student");
//
//    }
//
//    @Then("user enters password")
//    public void user_enters_password() {
//        loginPage.passwordTextInput.sendKeys("Bootcamp#5");
//
//    }

//    @When("user enters {string} username")
//    public void user_enters_username(String username) {
//        loginPage.usernameTextInput.sendKeys("yoll-student");
//
//    }
//    @Then("user enters {string} password")
//    public void user_enters_password(String password) {
//        loginPage.passwordTextInput.sendKeys("Bootcamp#5");
//
//    }
//
//    //=========================
//
//    @Then("user clicks on login button")
//    public void user_clicks_on_login_button() {
//        loginPage.loginButton.click();
//    }
    @When("user enters {string} username and {string} password and clicks
on login button")
    public void
user_enters_username_and_password_and_clicks_on_login_button(String
username, String password) {
        loginPage.login(username,password);
    }
    @Then("user validates {string} error message")
    public void user_validates_error_message(String errorMessage) {
       loginPage.verifyErrorMessage(errorMessage);
    }
```

```
//    @Then("user should be able to verify account name header on HRM
Homepage")
//    public void
user_should_be_able_to_verify_account_name_header_on_hrm_homepage() {
//        // Write code here that turns the phrase above into concrete
actions
//
//        String expectedHeader = "Welcome Yoll";
//        String actualHeader =
driver.findElement(By.id("welcome")).getText();
//        Assert.assertEquals("Account name header verification failed!",
expectedHeader, actualHeader);
//    }
    }
```

```
package utilities;
import io.github.bonigarcia.wdm.WebDriverManager;
import org.openqa.selenium. WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
public class DriverFactory {
    private static WebDriver driver;
    public static WebDriver getDriver () {
        if (driver == null) {
            WebDriverManager.chromedriver().setup();
            driver = new ChromeDriver();
        }
        return driver;
    }
        public static void closeDriver () {
            if (driver != null) {
                driver.close();
                driver = null;
            }
        }
    }
```

Singleton




```
package ui_automation.utilities;

import org.openqa.selenium.WebDriver;

public class Driver {

    private Driver(){}

    private static Driver driverInstance=new Driver();

    public static Driver getInstance(){
        return driverInstance;
    }

    ThreadLocal<WebDriver> threadLocalDriver=new ThreadLocal<WebDriver>();

    public WebDriver getDriver(){
        WebDriver driver = threadLocalDriver.get();
        return driver;
    }

    public  void setDriver(WebDriver driverParameter){
        // set the parameter driver into threadLocal
        threadLocalDriver.set(driverParameter);
    }

    public void removeDriver(){
        WebDriver driver = threadLocalDriver.get();
```

```
        driver.quit();
        threadLocalDriver.remove();
    }
}
```

This `Driver` class implements a **singleton design pattern** with a **thread-local storage mechanism** to manage WebDriver instances. Let's break it down with explanations of each part.

Purpose of the Class

1. **Singleton Pattern**: Ensures that there is only one instance of the `Driver` class throughout the application.
2. **ThreadLocal WebDriver**: Allows each thread to have its own instance of `WebDriver`. This is especially useful when running parallel tests to avoid conflicts between threads.

Code Explanation

1. Private Constructor

```
private Driver() {}
```

- Prevents instantiation of the `Driver` class from outside.
- Ensures only one instance is created using the `getInstance` method.

---

2. Singleton Instance

```
private static Driver driverInstance = new Driver();
```

- A static instance of the `Driver` class is created.
- This ensures the class follows the singleton pattern (only one instance is shared).

```
public static Driver getInstance() {
    return driverInstance;
}
```

- Provides a way to access the singleton instance.

---

3. ThreadLocal for WebDriver

```
ThreadLocal<WebDriver> threadLocalDriver = new ThreadLocal<>();
```

- `ThreadLocal` is a storage mechanism that ensures each thread has its own separate `WebDriver` instance.
- This is critical for parallel execution of tests, as each test needs its own independent browser session.

---

4. Get WebDriver

```
public WebDriver getDriver() {
    WebDriver driver = threadLocalDriver.get();
    return driver;
}
```

- Retrieves the `WebDriver` instance for the current thread.
- If no instance is set for the current thread, `null` is returned.

---

5. Set WebDriver

```
public void setDriver(WebDriver driverParameter) {
    threadLocalDriver.set(driverParameter);
}
```

- Sets the `WebDriver` instance for the current thread.
- The `driverParameter` is passed in from elsewhere in the code, typically after creating a browser session.

---

6. Remove WebDriver

```
public void removeDriver() {
    WebDriver driver = threadLocalDriver.get();
    driver.quit(); // Closes the browser session.
    threadLocalDriver.remove(); // Removes the WebDriver instance from the current
thread.
}
```

- Cleans up after a test by:
    1. Closing the browser session.
    2. Removing the `WebDriver` instance from the thread-local storage to free up resources.

Why Use This Class?

1. **Singleton Design Pattern**:
    - Ensures only one `Driver` instance is used throughout the test suite.
    - Simplifies management of driver configuration.
2. **ThreadLocal Storage**:

- Enables **parallel execution of tests** by providing isolated `WebDriver` instances for each thread.
- Prevents one thread's WebDriver instance from interfering with another.

Real-Life Use Case

When running parallel Selenium tests using a tool like TestNG or JUnit:

1. Each test will run in its own thread.
2. This `Driver` class ensures each thread gets its own browser instance.
3. At the end of each test, the browser is properly closed, and the thread-local WebDriver instance is removed.

This setup is highly scalable for running automated tests on multiple browsers in parallel, either locally or on a Selenium Grid.

Difference between Driver and DriverFactory classes in cucumber

The **`Driver`** and **`DriverFactory`** classes in a Cucumber-based Selenium framework serve a similar purpose—managing WebDriver instances—but they are implemented differently, often depending on the project's requirements. Here's a comparison to highlight their differences:

---

1. `Driver` Class

The `Driver` class, as in your example, typically uses a **singleton pattern** combined with **ThreadLocal storage** to manage WebDriver instances. It ensures each thread has its own WebDriver instance for parallel test execution.

Features of `Driver`:

- **Singleton Pattern**:
  - Ensures there is only one instance of the `Driver` class.
  - Provides a globally accessible `getInstance()` method.
- **ThreadLocal for Parallel Testing**:
  - Each thread (test execution unit) has its own `WebDriver` instance.
  - Prevents conflicts between threads during parallel execution.
- **Resource Cleanup**:
  - Includes methods like `removeDriver()` to clean up after tests.

Advantages of `Driver`:

- Ideal for **parallel execution** scenarios.
- Provides thread safety by isolating WebDriver instances across threads.
- Ensures efficient management of WebDriver lifecycle.

When to Use `Driver`:

- When your framework needs to support **parallel execution**.
- For advanced setups where each test thread requires an isolated WebDriver.

---

2. `DriverFactory` Class

The `DriverFactory` class is typically simpler and does not always implement ThreadLocal storage or a singleton pattern. It is often used to create and provide `WebDriver` instances for tests.

Features of `DriverFactory`:

- **Static Factory Methods**:
  - Contains a static `getDriver()` method to create or return a `WebDriver` instance.
- **Non-Thread-Safe**:
  - Does not isolate WebDriver instances between threads unless explicitly implemented.
- **Simple Lifecycle Management**:
  - Does not inherently manage the lifecycle of WebDriver instances across threads.

Advantages of `DriverFactory`:

- Simpler to implement and understand.
- Useful for single-threaded or non-parallel test setups.
- Can be extended to add browser-specific configurations.

When to Use `DriverFactory`:

- For basic test setups with no **parallel execution**.
- When simplicity is more important than thread safety.

Key Differences

Feature	Driver	DriverFactory
Pattern	Singleton + ThreadLocal	Factory Pattern
Parallel Testing Support	Yes (via ThreadLocal)	No, unless explicitly implemented
Complexity	More complex due to ThreadLocal usage	Simpler to understand and implement
Lifecycle Management	Manages lifecycle for each thread	Simple lifecycle management
Use Case	Parallel tests	Single-threaded tests

Real-Life Example

1. Using `Driver`:

- You are running parallel Selenium tests on a Selenium Grid or a cloud platform (e.g., BrowserStack).
- Each test thread needs its own browser instance without interference from others.

2. Using `DriverFactory`:

- You are running tests sequentially in a single thread.
- Simplicity is more important, and parallel execution is not a requirement.

In summary:

- **Use `Driver`** when building a scalable, parallel-test-ready framework.
- **Use `DriverFactory`** for simpler test setups or when parallel execution is not required.

HOOKS

```
import io.cucumber.core.api.Scenario;
import io.cucumber.java.After;
import io.cucumber.java.Before;
import org.openqa.selenium.OutputType;
import org.openqa.selenium.TakesScreenshot;
import org.openqa.selenium.WebDriver;
import ui_automation.utilities.BrowserFactory;
import ui_automation.utilities.Driver;

import java.util.concurrent.TimeUnit;

public class Hooks {
    public WebDriver driver=null;

    @Before
    public void setUp(){
        driver= BrowserFactory.createInstance();
        Driver.getInstance().setDriver(driver);
        driver = Driver.getInstance().getDriver();
        driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
        //driver.manage().window().maximize();
    }

    @After
    public void tearDown(Scenario scenario) {
        if (scenario.isFailed()) {
            final byte[] screenshot = ((TakesScreenshot)
Driver.getInstance().getDriver()).getScreenshotAs(OutputType.BYTES);
            scenario.embed(screenshot, "image/png");
        }
       Driver.getInstance().removeDriver();
```

```
    }
}
```

Hooks in Cucumber

Hooks in Cucumber are special blocks of code that can run **before** or **after** each scenario. They are primarily used for setup and teardown activities to manage preconditions and post-test actions. Hooks are annotated with:

- **`@Before`**: Runs before each scenario.
- **`@After`**: Runs after each scenario.

This helps to avoid repetitive code in step definitions, making the framework cleaner and more maintainable.

Explanation of the Code

Class: `Hooks`

The `Hooks` class contains two methods (`setUp` and `tearDown`) annotated with `@Before` and `@After`, respectively, to handle setup and teardown for each test scenario.

---

1. `@Before` Method: `setUp()`

This method runs before each scenario starts.

1. **`driver = BrowserFactory.createInstance();`**
   - Creates a new WebDriver instance using `BrowserFactory`. This can include browser-specific configurations like initializing Chrome, Firefox, etc.
2. **`Driver.getInstance().setDriver(driver);`**
   - Sets the created WebDriver instance into the `Driver` class using its `ThreadLocal` mechanism.
   - This ensures thread safety during parallel test execution.
3. **`driver = Driver.getInstance().getDriver();`**
   - Retrieves the thread-safe WebDriver instance from the `Driver` class.
4. **`driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);`**
   - Configures an implicit wait of 10 seconds, allowing elements some time to appear before throwing an exception.
5. *(Optional)* **`//driver.manage().window().maximize();`**
   - This can be uncommented to maximize the browser window before starting a test.

---

2. `@After` Method: `tearDown(Scenario scenario)`

This method runs after each scenario ends.

1. **`if (scenario.isFailed())`**
   - Checks if the scenario failed.
   - If true, it takes a screenshot to help debug the failure.
2. **`final byte[] screenshot = ((TakesScreenshot) Driver.getInstance().getDriver()).getScreenshotAs(OutputType.BYTES);`**
   - Captures a screenshot of the current browser state as a byte array.
3. **`scenario.embed(screenshot, "image/png");`**
   - Embeds the screenshot in the Cucumber report for better visibility of what went wrong.
4. **`Driver.getInstance().removeDriver();`**
   - Quits the WebDriver instance.
   - Removes the WebDriver from `ThreadLocal` to clean up resources and prevent memory leaks.

How It Works in a Test Workflow

1. **Before the Scenario**:
   - A WebDriver instance is created and configured.
   - This WebDriver instance is stored in a thread-safe `Driver` class, ready to be used by the step definitions.
2. **During the Scenario**:
   - The WebDriver instance is used to perform browser actions as defined in the step definitions.
3. **After the Scenario**:
   - If the scenario fails, a screenshot is taken and embedded in the report.
   - The WebDriver instance is closed, and its reference is removed from memory.

Key Benefits

- **Reusable Setup/Teardown**: Ensures all scenarios have consistent setup and teardown procedures without duplicating code.
- **Thread Safety**: Works seamlessly in parallel execution because of the `Driver` class and `ThreadLocal` usage.
- **Debugging Aid**: Captures screenshots on failures, which helps in diagnosing issues.

Real-Life Example

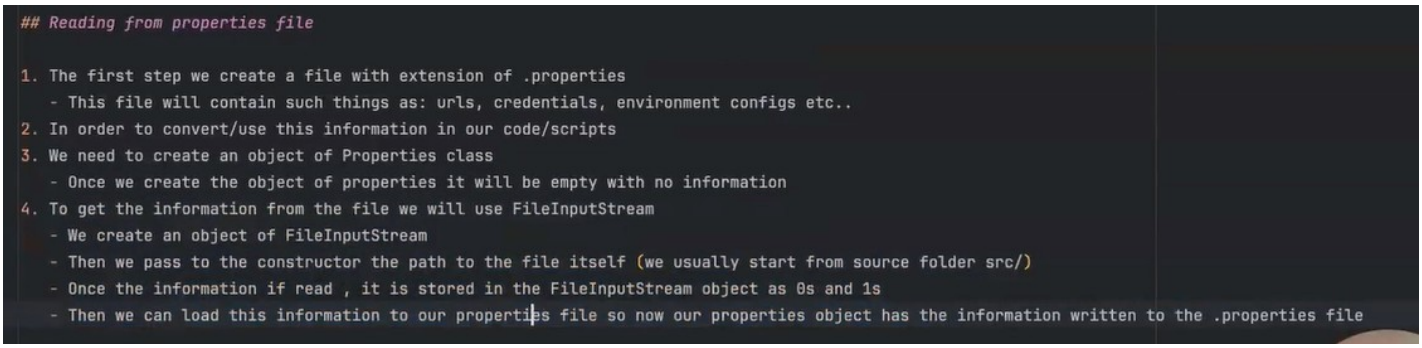
Scenario:

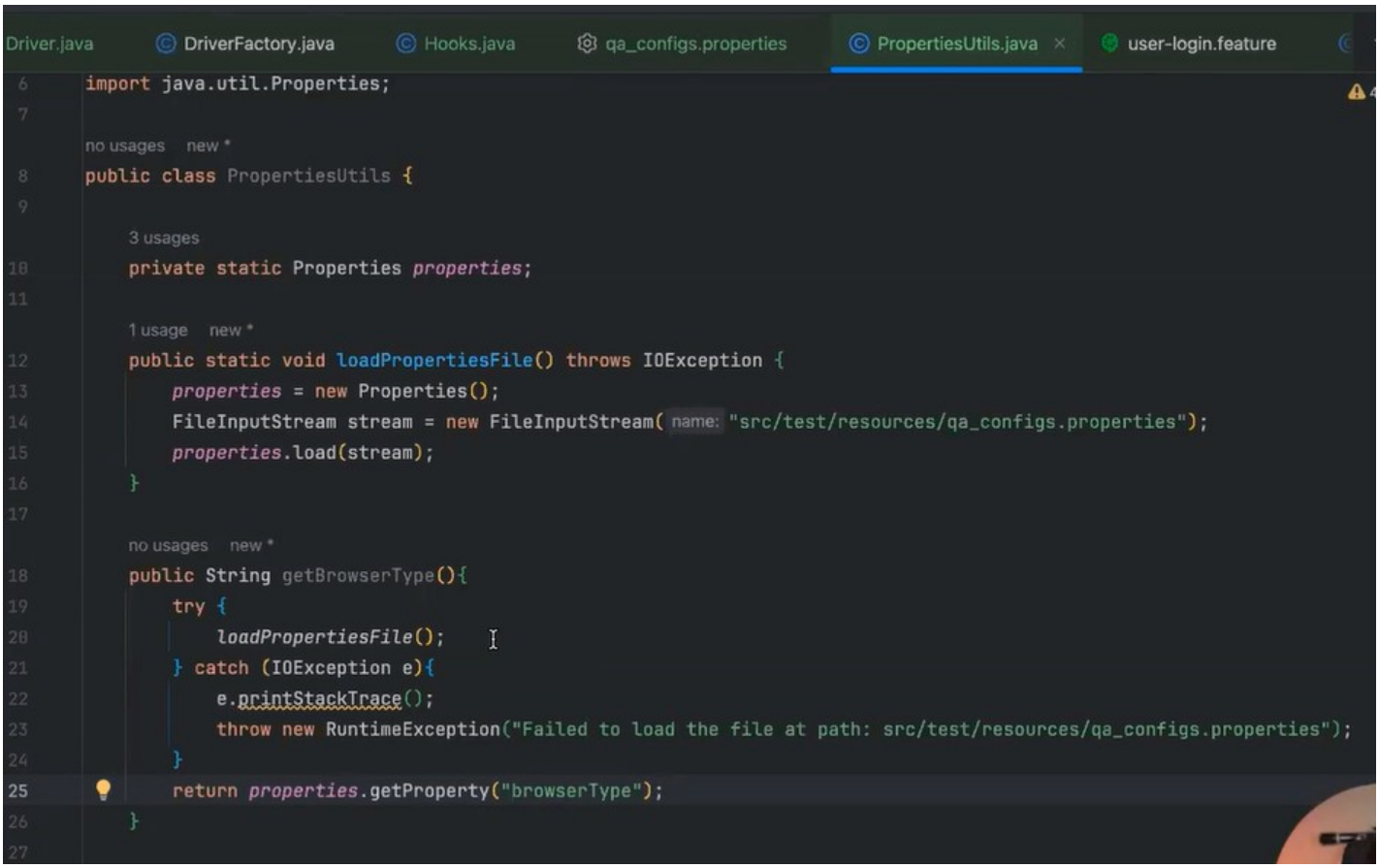
Testing the login functionality of a web application.

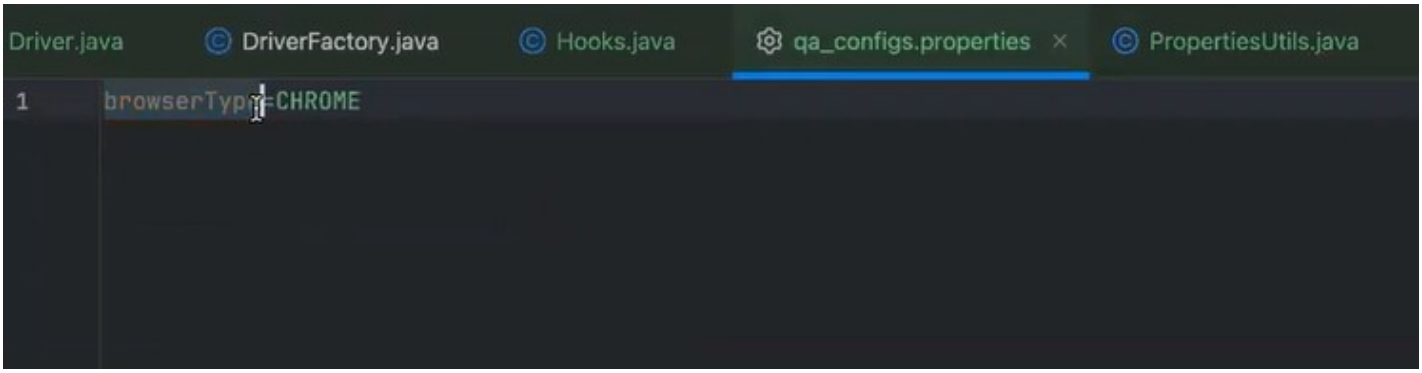
1. **Before the test:**
   - Opens the browser, navigates to the application URL, and sets implicit wait.
2. **Test Execution:**
   - Executes steps like entering credentials and verifying the result.
3. **After the test:**
   - If login fails, captures a screenshot and embeds it in the report.
   - Closes the browser and cleans up resources.

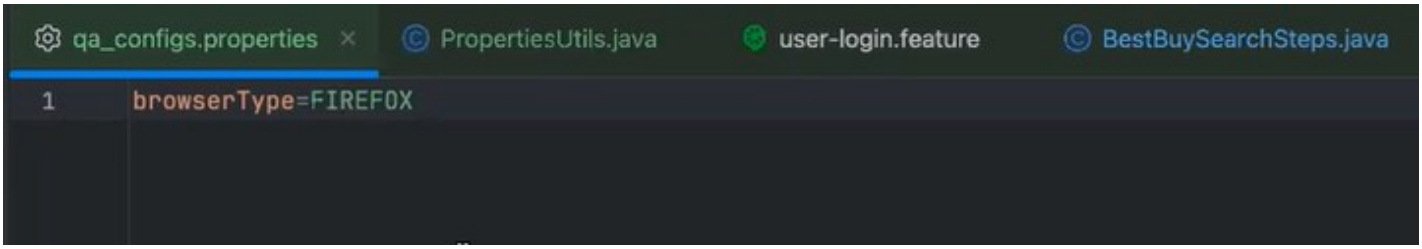
This approach ensures all tests are executed under similar conditions and are resource-efficient.

Properties Class and qa-configs.properties files

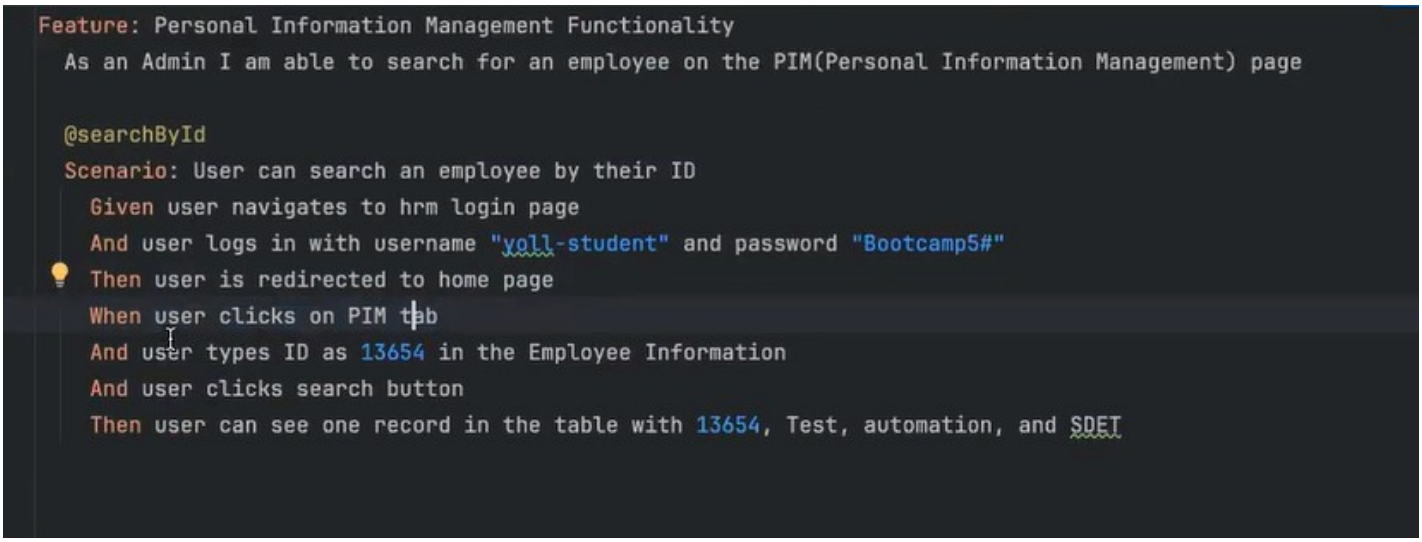

Reading from properties file
1. The first step we create a file with extension of .properties
- This file will contain such things as: urls, credentials, environment configs etc..
2. In order to convert/use this information in our code/scripts
3. We need to create an object of Properties class
- Once we create the object of properties it will be empty with no information
4. To get the information from the file we will use FileInputStream
- We create an object of FileInputStream
- Then we pass to the constructor the path to the file itself (we usually start from source folder src/)
- Once the information if read , it is stored in the FileInputStream object as 0s and 1s
- Then we can load this information to our properties file so now our properties object has the information written to the .properties file







Pim-page.feature


Feature: Personal Information Management Functionality
As an Admin I am able to search for an employee on the PIM(Personal Information Management) page
@searchById
Scenario: User can search an employee by their ID
Given user navigates to hrm login page
And user logs in with username "yoll-student" and password "Bootcamp5#"
Then user is redirected to home page
When user clicks on PIM tab
And user types ID as 13654 in the Employee Information
And user clicks search button
Then user can see one record in the table with 13654, Test, automation, and SDET